Automatic Datapath Abstraction of Pipelined Circuits

by

Vlad Ciubotariu

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Abstract

Pipelined circuits operate as an assembly line that starts processing new instructions while older ones continue execution. Control properties specify the correct behaviour of the pipeline with respect to how it handles the concurrency between instructions. Control properties stand out as one of the most challenging aspects of pipelined circuit verification. Their verification depends on the datapath and memories, which in practice account for the largest part of the state space of the circuit. To alleviate the state explosion problem, abstraction of memories and datapath becomes mandatory. This thesis provides a methodology for an efficient abstraction of the datapath under all possible control-visible behaviours. For verification of control properties, the abstracted datapath is then substituted in place of the original one and the control circuitry is left unchanged. With respect to control properties, the abstraction is shown conservative by both language containment and simulation.

For verification of control properties, the pipeline datapath is represented by a network of registers, unrestricted combinational datapath blocks and muxes. The values flowing through the datapath are called parcels. The control is the state machine that steers the parcels through the network. As parcels travel through the pipeline, they undergo transformations through the datapath blocks. The control-visible results of these transformations fan-out into control variables which in turn influence the next stage the parcels are transferred to by the control. The semantics of the datapath is formalized as a labelled transition system called a parcel automaton. Parcel automata capture the set of all control visible paths through the pipeline and are derived without the need of reachability analysis of the original pipeline. Datapath abstraction is defined using familiar concepts such as language containment or simulation. We have proved results that show that datapath abstraction leads to pipeline abstraction.

Our approach has been incorporated into a practical algorithm that yields directly the abstract parcel automaton, bypassing the construction of the concrete parcel automaton. The algorithm uses a SAT solver to generate incrementally all possible control visible behaviours of the pipeline datapath. Our largest case study is a 32-bit two-wide superscalar OpenRISC microprocessor written in VHDL, where it reduced the size of the implementation from 35k gates to 2k gates in less than 10 minutes while using less than 52MB of memory.

Acknowledgments

I am very grateful to my supervisor Mark Aaagaard without whose support and encouragements this thesis would not have reached completion. Also, I am grateful for all the good friends I have been blessed with, and I would like to thank my friend Marjan for her comforting help during the hardest moments.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Hardware and software systems have a pervasive presence in our day to day lives. From mass produced computer chips and embedded software in our computing devices and cars to systems that control high speed trains, aeroplanes and nuclear plants, our well being and safety is increasingly dependent on whether such systems behave as intended. The consequences of malfunctioning hardware and software range from the nuisance of continuous operating system updates and firmware for our computers and gadgets to mass recalls with high economic cost in the order of millions and billions of dollars and catastrophic accidents that put human life at risk.

The traditional approach to validate hardware and software systems is testing. The testing paradigm uses input vectors to check the input-output behaviour of the system or to drive the system through execution scenarios which are checked against the expected behaviour. Black box testing or functional testing checks the input-output behaviour of the system and is oblivious to its internal structure. White box testing uses the structural information about the system, such as the control flow graph of the program code, to craft input vectors that drive it through execution paths that visit the structural elements such as lines of code or control-flow conditions. The success of testing is measured using coverage metrics.

For medium to large systems the sheer multitude of possible input values makes an exhaustive approach intractable. Thus complete validation through testing is not achievable except for small scale systems. Even when complete coverage is achieved it is not a certainty that all system behaviours have been exercised, since coverage metrics are defined in terms of the structure of the system while there can be exponentially more behaviours. Achieving reasonable coverage through testing in modern day microprocessors takes enormous amounts of time. For instance, simulating a few minutes of a 1GHZ microprocessor takes almost 6 months of simulation time on a large cluster of workstations [Bentley, 2001].

Formal methods [Clarke and Wing, 1996, Clarke and Kurshan, 1996, Dill and Rushby, 1996, Hall,

1990] stand for the collection of methods that apply mathematical reasoning to the proof of correctness of hardware and software systems. The application of formal methods to a system is called formal verification. Formal verification checks a given property holds of all behaviours of the system and is therefore exhaustive, providing a definitive answer to correctness. Formal verification has been applied successful to a wide range of hardware and software systems [Bentley, 2001, Clarke et al., 1995, McMillan, 2001, Dill et al., 1992]. In the real world, formal verification and testing coexist and techniques from formal verification have been used achieve better testing, creating hybrid methodologies.

Due to the theoretical complexity of program verification, ranging from NP-hard to undecidable, with formal verification comes the tradeoff between automation and capacity. In the wide spectrum of formal methods we distinguish two categories of techniques. At one end we have deductive methods that provide virtually unbounded capacity, being able to verify infinite systems, but are also more likely to rely on the intervention of a knowledgeable user to guide the proof. At the other end we have algorithmic methods that are highly automated but are not directly applicable to large systems. The gap between the two types of formal methods is bridged using abstraction and decomposition techniques.

In this thesis we are concerned with using abstraction to improve the capacity of one such automated technique, called model checking, for the verification of pipelined circuits. In model checking, the verified system is called a reactive system and the language in which the properties are described is called temporal logic. To verify properties, model checking performs an exhaustive search of the state space of the reactive system. The size of the state space is the major challenge that impedes the direct use of model checking. The capacity problem incurred due to the state space factor is called the state explosion problem. The state space explosion problem is mitigated using abstraction and decomposition. Our research is concerned with abstraction in the application domain of pipelined circuits.

Pipelining uses the same principle as an assembly line that shifts products simultaneously from one assembly stage to the next. A pipelined circuit divides the execution of instructions, also called parcels, into stages. Upon entering the pipeline, the execution of an instruction happens incrementally as it moves from one stage to the next, until it exits the pipeline. To achieve a similar productivity increase to the assembly line, the pipeline overlaps the execution of instructions whereby each execution stage holds a different instruction. Compared to a non-pipelined circuit that performs the same operation, in an ideal linear pipeline that does not have instruction dependencies, the execution time of individual instructions does not change. However the number of instructions processed per unit of time increases proportionally to the length of the pipeline. The theoretical speedup of a pipeline is not achieved in practice due to dependencies between instructions [Hartstein and Puzak, 2002].

A pipelined circuit consists of a network of stages through which the parcels flow, memories and register files that store instructions and the data operated on, datapath elements that perform the operation corresponding to each pipeline stage and control circuitry that orchestrates the execution of the instructions. The network of stages is linear but only in the simplest circuits. It may contain branches and loops and the paths taken by instructions may be selected dynamically by the control based on the current execution context. The concurrent execution of instructions leads to possible race conditions which in pipeline circuits are called hazards.

There are three types of hazards. Data hazards arise due to data dependencies when the operand of one instruction, the consumer, is created by another, the producer. In such cases, the consumer waits — it stalls — until the operand is ready. Structural hazards arise from resource contention when multiple instructions need to transfer to the same next stage. Finally, control hazards happen due to speculative execution. Instructions that were fetched after a control instruction that is mispredicted, must be removed from the pipeline.

Because of the synchronization problems it solves, the control circuitry is the main source of complexity in the pipeline circuit and therefore, the most likely part of the design to contain design errors. The size of the controller is often within the verification capacity of model checking techniques. What prevents its direct verification is the size of the memories and datapaths which contribute the largest proportion to the state space of the circuit. Memories are easier to abstract, their size reduces to what is needed to accommodate the read and write locations of the maximum number of in-flight instructions. Datapaths are more challenging to abstract because they use the parcel's value to generate feedback signals to the control circuitry and thus affect the overall execution in the pipeline.

In this thesis we present a novel datapath abstraction technique. The methodology is described pictorially in Figure 1.1. At the core of our approach is the use of a mathematical representation, called parcel automata, to describe the control-visible behaviour of the parcels as they travel through the pipeline. A parcel's behaviour is defined by both the control signals it generates at each stage in the pipeline and the path it takes through the pipeline. In our methodology datapath abstraction is performed by abstracting the concrete parcel automata and then using the abstract parcel automata to define abstract datapaths that are then substituted in place of the concrete ones. The process of replacing the concrete datapath by an abstract one is a form of abstract interpretation. We show that the conventional forms to define abstraction of automata, such as simulation and language containment, carry over to abstract interpretation of the pipeline datapath using parcel automata.

Our contribution is threefold. First, we contribute a formal framework for datapath abstraction using parcel automata. Within this framework we define pipeline models and parcel automata, abstraction of parcel automata and prove correctness of pipeline abstraction using abstract parcel automata. Second, we provide an abstraction algorithm for parcel automata. Our algorithm tackles the state

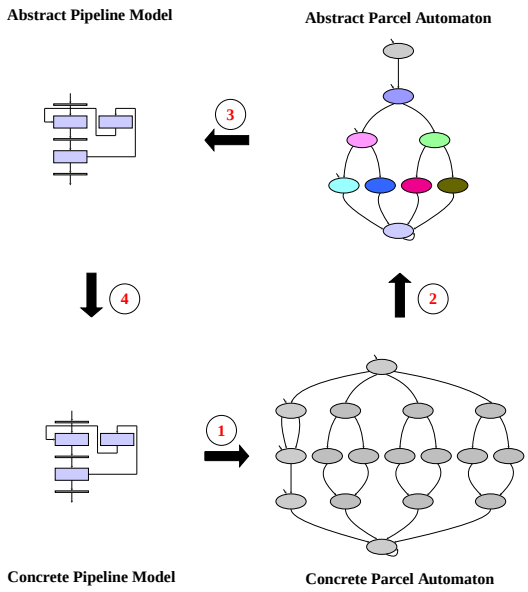


Figure 1.1: Overview of abstraction methodology.

explosion problem by the symbolic traversal of the concrete parcel automaton using a satisfiability solver. And third, our approach is implemented in a prototype. The tool allows its users to create pipeline models by specifying the top level structure of interconnected pipeline stages. The datapath and control are defined directly in VHDL and then referenced from the high-level model. Datapath abstraction is then performed with the click of a button. As cases studies we used several designs, spanning from simple 32-bit arithmetic pipelines to an edge detector circuit and a 32-bit superscalar OpenRISC microprocessor.

1.1 Overview Of Background And Related Work

The most complex pipelined circuits appear in today's microprocessors. Approaches to formal verification of microprocessors include deductive methods using theorem proving and automated techniques using model checking or specialized decision procedures. With any such verification technique one must specify the correctness criteria and then provide a sound verification strategy.

Often, the correctness statement specifies a relationship described using concepts from automata theory. Simulation states that a relationship between the value of specification variables in the implementation states and the specification states is preserved after an execution step of the implementation and specification. Another way to define correctness is through language containment. In this definition, the sets of implementation traces are compared against the set of traces of the

specification. Formulating correctness statements for pipelined circuits is made challenging by the fact that the implementation variables that correspond to the specification reflect changes by partially executed instructions. The pipelined circuit executes multiple instructions in one step while the specification is sequential and executes one instruction at a time.

A correctness framework that overcomes the challenges of aligning the implementation and specification states is called flushing [Burch and Dill, 1994]. In this approach, the implementation state is run, without fetching new instructions into the pipeline, until all the instructions being currently executed complete. Verification techniques using flushing have been performed using both deductive and automatic methods. In deductive approaches, the challenge of applying flushing is identifying proof strategies to deal with the various optimizations of a microprocessor design such as scoreboarding, execution units with variable latency and branch prediction [Sawada and Hunt, 1997, Hosabettu et al., 1998, Skakkebak et al., 1998]. Automatic techniques use efficient decision procedures to prove a simulation relation based on flushing [Lahiri et al., 2002, Velev and Bryant, 2000, Manolios and Srinivasan, 2004].

In a hazard based approach [Aagaard, 2003] the top-level correctness is defined with respect to the three types of pipeline hazards: data, structural and control. Hazard correctness is formulated in terms of pipeline specific properties and thus are more straightforward to define. Most of these properties target the control circuitry. The main impediment in the automatic verification of control properties of pipelined circuits is the large contribution of datapath and memories to their overall size.

Approaches to datapath abstraction vary in the degree of automation and precision and often perform a tradeoff. When a decision procedure is used, datapath abstraction is performed using uninterpreted functions. In model checking, datapath abstraction is done by reducing the bitwidth of the operands. The equivalent of uninterpreted functions in this context is to sever the feedback signals from datapath to control and replace the datapath implementation by wires [Ho et al., 1998]. However, imprecise abstractions pose the threat of false counterexamples: traces of the abstract circuit that violate the property and do not have an equivalent trace in the concrete implementation. The solution is to refine the abstraction in a refinement loop [Andraus et al., 2006] until the property passes or a true counterexample is found.

1.2 Approach And Contributions

Our approach to datapath abstraction targets the verification of control properties. We exploit structural rules in the design of pipelined circuits to derive efficient and accurate abstractions. Our approach is particularly useful for properties that specify the parcel flow through the pipeline and are sensitive to the latency of the paths through the pipeline.

Our contribution is an abstraction technique that leverages a novel mathematical representation for the pipeline datapath using a type of automata, called parcel automata. A parcel automaton is a mathematical model for the execution of an instruction. Each state of the automaton represents a parcel in a pipeline state. A transition denotes the transformation of the parcel by the datapath as it moves to the next stage. The label of the transition indicates the control visible effects. A run of the parcel automaton corresponds to a run of a single parcel through the pipeline.

Formalizing the datapath as an automaton presents the advantage of clean mathematical reasoning about datapath abstraction. Simulation and language containment formalize the notion of equivalent parcels, parcels that have the same control visible behaviour as they move through the pipeline. We prove in our framework that both simulation and language containment on parcel automata transfer to pipeline abstraction using parcel automata. Our abstraction algorithm uses a symbolic method based on SAT to simultaneously traverse all equivalent runs of the parcel automaton. The abstraction of the pipeline datapath reduces to collapsing the equivalent runs of the concrete parcel automaton into a run of the abstract parcel automaton. Datapath abstraction is represented as abstraction of parcel automata and pipeline abstraction for control properties is performed as a form of abstract interpretation using abstract datapaths derived from abstract parcel automata.

We have implemented our methodology as part of a verification flow using a prototype tool called Bluenose II. The tool reads the annotated model that describes the structure of the pipeline as a network of interconnected stages. The descriptions of the stages reference the VHDL files that implement the datapaths. Datapath abstraction using a SAT solver is performed by generating CNF formulas from the netlists obtained by synthesis of the datapath files. From the abstract parcel automaton the tool generates VHDL for the abstract datapaths which in turn define the abstract pipeline circuit through abstract interpretation. The abstract circuit is then verified with a model checker.

1.3 Outline Of Thesis

Chapter 2 introduces known concepts that we use throughout the thesis: labeled state machines, circuits and model checking. It also discusses related work. Chapter 3 defines the model of pipelined circuits. In Chapter 4 we present the definition of parcel automata, their abstraction and use in abstract interpretation of pipelined circuits. Chapter 5 describes parcel maps and the correctness of datapath abstraction using parcel automata. Chapter 6 defines path abstraction for parcel automata and presents abstraction algorithms and case studies. Chapter 7 is a summary of the thesis and of our contributions.

Chapter 2

Background And Related Work

In this chapter we describe several concepts that are used throughout the thesis: labeled transition system, model checking and circuits. In Section 2.4 we present related work.

2.1 Labeled Transition Systems

2.1.1 Definition (Labeled Transition System)**.** A labeled transition system is a tuple $M = \langle Q, R, T, I \rangle$:

- Q denotes the set of states
- T is the set of transition labels
- $R \subseteq Q \times T \times Q$ is the transition relation
- $I \subseteq Q$ is the set of initial states

The language $\mathcal{L}(M)$ of a labeled transition M is the set of infinite runs represented by functions of form $run : \mathbf{N} \to Q \times T$ such that $run\ k = (q^k, t^k)$ and $q^0 \in I \wedge \forall\, k.\ (q^k, t^k, q^{k+1}) \in R$.

Consider two labeled transition systems $M_1 = \langle Q_1, R_1, T_1, I_1 \rangle$ and $M_2 = \langle Q_2, R_2, T_2, I_2 \rangle$. In order to formulate criteria of abstraction, we need to compare states and transition labels of the two transition systems. In general they may belong to disjoint sets and therefore may not be directly comparable. Instead of direct equality, for states we use the labeling functions $lab_{Q1} : Q_1 \to L_Q$ and $lab_{Q2} : Q_2 \to L_Q$ with the same codomain L_Q, and, respectively, for transitions use $lab_{T1} : T_1 \to L_T$ and $lab_{T2} : T_2 \to L_T$, sharing the codomain L_T.

Abstraction of labeled transition systems is formulated using simulation [Milner, 1971] or language containment.

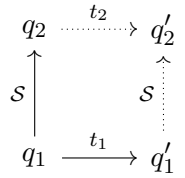


Figure 2.1: Commuting diagram for the simulation relation $\mathcal{S}$.

2.1.2 Definition (Simulation)**.** A simulation relation between M_1 and M_2 is a relation $\mathcal{S} \subseteq Q_1 \times Q_2$ that satisfies the following conditions:

1. $\mathcal{S}$ is compatible with the state labeling.

$$\forall\, q_1 \in Q_1.\, \forall\, q_2 \in Q_2.\, (q_1, q_2) \in \mathcal{S} \Longrightarrow lab_{Q1}\, q_1 = lab_{Q2}\, q_2$$

2. $\mathcal{S}$ is total over the initial states of M_1.

$$\forall\, q_1 \in I_1.\, \exists\, q_2 \in I_2.\, (q_1, q_2) \in \mathcal{S}$$

3. $\mathcal{S}$ is invariant under the transition relations.

$$\begin{array}{l} \forall\, q_1 \in Q_1.\, \forall\, t_1 \in T_1.\, \forall\, q_1' \in Q_1. \\ \quad \forall\, q_2 \in Q_2. \\ \qquad (q_1, q_2) \in \mathcal{S} \wedge (q_1, t_1, q_1') \in R_1 \Longrightarrow \\ \qquad\quad \exists\, t_2 \in T_2.\, \exists\, q_2' \in Q_2.\, lab_{T1}\, t_1 = lab_{T2}\, t_2 \wedge (q_2, t_2, q_2') \in R_2 \wedge (q_1', q_2') \in \mathcal{S} \end{array} \tag{2.1}$$

Pictorially, Equation 2.1 is described in Figure 2.1. We say that M_2 simulates M_1 if there exists a simulation relation $\mathcal{S} \subseteq Q_1 \times Q_2$.

The other way to state abstraction is through language containment. Language containment between M_1 and M_2 holds if for every $run_1 \in \mathcal{L}(M_1)$ there exists an equivalent run $run_2 \in \mathcal{L}(M_2)$:

$$\begin{array}{l} \forall\, run_1 \in \mathcal{L}(M_1).\, \exists\, run_2 \in \mathcal{L}(M_2). \\ \quad \forall\, k \in \mathbf{N}.\, lab_{Q1}\, q_1^k = lab_{Q2}\, q_2^k \wedge lab_{T1}\, t_1^k = lab_{T2}\, t_2^k \end{array} \tag{2.2}$$

2.1.3 Proposition. Simulation implies language containment.

Proof. Let $\mathcal{S}$ be a simulation relation between M_1 and M_2. Consider $run_1 \in \mathcal{L}(M_1)$. We construct

by induction the sequence $q_2^0, t_2^0, q_2^1, \ldots, q_2^n$ such that $(q_1^n, q_2^n) \in \mathcal{S}$ and for all $k \leq n-1$

$$lab_{Q1}\, q_1^k = lab_{Q2}\, q_2^k \tag{2.3}$$

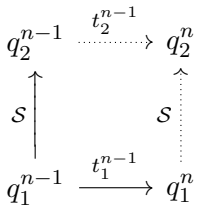
$$lab_{T1}\, t_1^k = lab_{T2}\, t_2^k \tag{2.4}$$

$$(q_2^k, t_2^k, q_2^{k+1}) \in R_2 \tag{2.5}$$

If $q_2^0 \in I_2$ and Equation 2.5 holds for $k \in \mathbf{N}$ then the function $run_2 : \mathbf{N} \to Q_2 \times T_2$ defined by $run_2\, k = (q_2^k, t_2^k)$ is a run of M_2 equivalent to run_1.

Base Case $n = 0$. We choose q_2^0 such that $(q_1^0, q_2^0) \in \mathcal{S}$.

Inductive Case $n > 0$ By induction we have that $(q_1^{n-1}, q_2^{n-1}) \in \mathcal{S}$. Therefore, there exist $t_2^{n-1} \in T_2$ and $q_2^n \in Q_2$ that make the diagram below commute:



□

The converse of Proposition 2.1.3 is not necessarily true as shown by the example in Figure 2.2. For each of the two possible runs of M_1 there exists an equivalent run of M_2. However, there is no state of M_2 that simulates the initial state q_{10}.

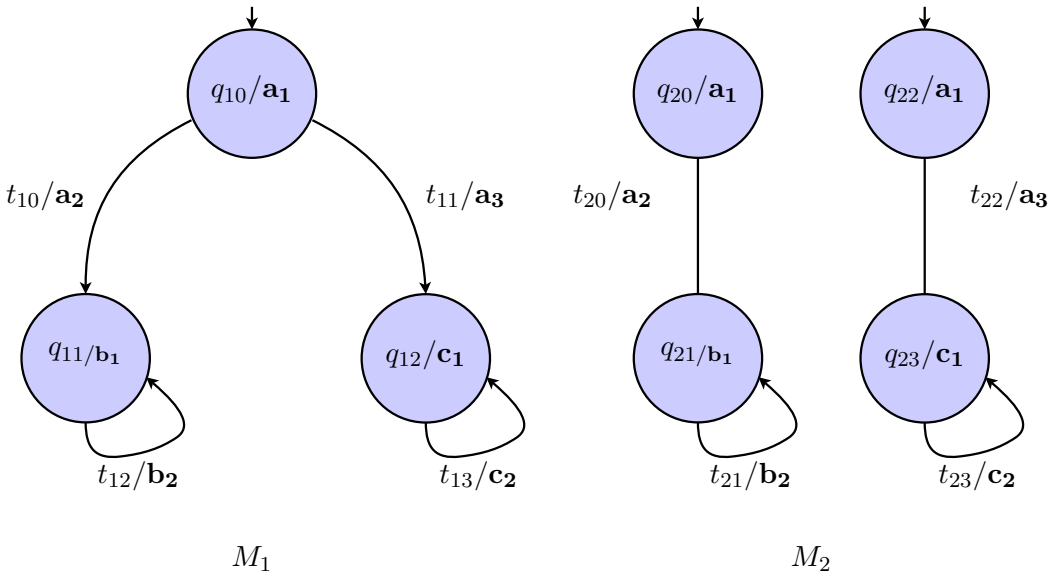


Fig. 2.2a. M_1 and M_2.

state	label	transition	label
q_{10}	$\mathbf{a_1}$	t_{10}	$\mathbf{a_2}$
q_{11}	$\mathbf{b_1}$	t_{11}	$\mathbf{a_3}$
q_{12}	$\mathbf{c_1}$	t_{12}	$\mathbf{b_2}$
		t_{13}	$\mathbf{c_2}$

Fig. 2.2b. Labeling of M_1.

state	label	transition	label
q_{20}	$\mathbf{a_1}$	t_{20}	$\mathbf{a_2}$
q_{21}	$\mathbf{b_1}$	t_{21}	$\mathbf{b_2}$
q_{22}	$\mathbf{a_1}$	t_{22}	$\mathbf{a_3}$
q_{23}	$\mathbf{c_1}$	t_{23}	$\mathbf{c_2}$

Fig. 2.2c. Labeling of M_2.

Figure 2.2: Example showing language containment without simulation.

2.2 Model Checking

Traditional approaches to verification use deductive methods and focus mainly on the proof of correctness of sequential systems [Hoare, 1969, Lamport, 1980]. Such techniques rely on precondition and postcondition properties of smaller programs to derive correctness properties of larger programs. Model checking uses temporal logic [Pnueli, 1977, Clarke et al., 1986] to specify properties about systems that run on an ongoing basis and automated techniques to verify such properties.

There are two main approaches in model checking. In the automata theoretic approach [Vardi, 1996], the temporal specification is converted to an automaton, negated and then intersected with the automaton for the implementation. The property holds if the language of the intersection is empty. In the algorithmic approach, the specification is verified directly by graph traversal on the automaton that represents the implementation. The implementation automaton can be represented either explicitly by storing its states in memory or symbolically using Boolean functions that represent their corresponding characteristic function. The latter form is called symbolic model checking [Burch et al., 1992] and the Boolean functions are represented compactly using binary decision diagrams [Bryant, 1986].

We present a temporal logic called $\mathbf{CTL}^*$ [Clarke et al., 1986]. The semantics of the logic is described in terms of a state machine that represents the implementation, called a Kripke structure. A Kripke structure is a tuple $M = \langle Q, R, I, AP, L \rangle$ where $\langle Q, R, I \rangle$ represents the state machine and $L : Q \to 2^{AP}$ is a labeling function that describes the set of atomic propositions that hold in a given state.

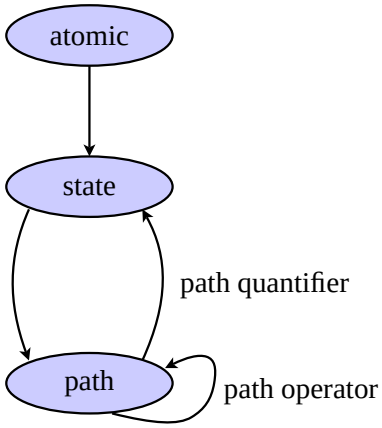


Figure 2.3: Structure of $\mathbf{CTL}^*$ formulas.

The structure of $\mathbf{CTL}^*$ formulas is described in Figure 2.3. $\mathbf{CTL}^*$ has two types of temporal formulas. Atomic formulas are those in the set AP. Any atomic formula is also a state formula,

and any state formula is also a path formula, verified on the first state of the path. Complex path formulas are constructed using path operators. The application of a path quantifier to a path formula yields a state formula. The syntax of formulas is defined inductively as follows:

Atomic Formulas If $p \in AP$ is a proposition then p is an atomic formula.

State Formulas

- Any atomic formula is also a state formula.
- If f is a path formula then $\mathbf{E}\ f$ and $\mathbf{A}\ f$ are state formulas.

Path Formulas

- Any state formula is also a path formula.
- If f is a path formula then $\mathbf{X}\ f$, $\mathbf{G}\ f$, $\mathbf{F}\ f$ are paths formulas.
- If f and g are path formulas then $f\ \mathbf{U}\ g$ is a path formula.

$\mathbf{E}$ and $\mathbf{A}$ are called path quantifiers. $\mathbf{E}\ f$ holds in a state if there exists a path from the state that satisfies f. Similarly, $\mathbf{A}\ f$ holds in a state, if for all paths from the state f holds. The formula $\mathbf{X}\ f$ is true of a path if f holds on the path starting in the next state. $\mathbf{G}\ f$ holds when f holds on all suffixes of the path, i.e. holds globally. Similarly, $\mathbf{F}\ f$ holds if there exists a suffix that satisfies f, i.e. f holds in the future. The formula $f\ \mathbf{U}\ g$ states that f must hold continously on all suffixes of the path up to, but not necessarily including, the suffix where g holds.

The transition relation of a Kripke structure is required to be total, i.e. for any state q there exists a state q' such that $(q, q') \in R$. If π is an infinite path $(q^0, q^1, \ldots)$ we denote the suffix that starts at position k by $\pi^k = (q^k, q^{k+1}, \ldots)$. Satisfiability of $\mathbf{CTL}^*$ formulas is defined as follows:

Atomic Formulas

$$q \models p \equiv p \in L(q)$$

State Formulas

$$\begin{aligned} q \models \mathbf{E}\ f &\equiv \exists\, \pi = (q^0, q^1, \ldots).\ q = q^0 \wedge \pi \models f \\ q \models \mathbf{A}\ f &\equiv \forall\, \pi = (q^0, q^1, \ldots).\ q = q^0 \Longrightarrow \pi \models f \end{aligned}$$

Path Formulas

$$\begin{aligned} \pi \models \mathbf{X}\ f &\equiv \pi^1 \models f \\ \pi \models \mathbf{G}\ f &\equiv \forall\, k.\ \pi^k \models f \\ \pi \models \mathbf{F}\ f &\equiv \exists\, k \in \mathbf{N}.\ \pi^k \models f \\ \pi \models f\ \mathbf{U}\ g &\equiv \exists\, k.\ \pi^k \models g \wedge \forall\, l < k.\ \pi^l \models f \end{aligned}$$

There are two subsets of $\mathbf{CTL}^*$ that are used in practice. $\mathbf{LTL}$ is the subset that consists of formulas of form $\mathbf{A}\ f$ where f is a path formula that does not contain path quantifiers. The other subset is $\mathbf{CTL}$ which allows only for formulas in which the occurrence of a path operator is preceded by a path quantifier. The $\mathbf{CTL}$ temporal operators become $\mathbf{EX}$, $\mathbf{EG}$, $\mathbf{EF}$, $\mathbf{EU}$ and respectively $\mathbf{AX}$, $\mathbf{AG}$, $\mathbf{AF}$, $\mathbf{AU}$.

$\mathbf{ACTL}^*$ is a subset of $\mathbf{CTL}^*$ that does not include existential quantifiers. The temporal properties of $\mathbf{ACTL}^*$ are preserved by simulation. Since $\mathbf{LTL}$ is a subset of $\mathbf{ACTL}^*$, simulation also preserves $\mathbf{LTL}$ properties. In addition, $\mathbf{LTL}$ is also preserved by language containment.

2.3 Circuits

In this section we formalize a language for describing circuits. The syntax and semantics of the language are similar to the ones provided by model checkers such as NuSMV [Cimatti et al., 2002]. Circuits are defined using a language for bitvector expressions and their semantics is given using labeled transition systems.

A name is a string of characters that begins with a letter or underscore and then continues with zero or more letters, underscore or digits. An identifier is a sequence of one or more names separated by a '.'. Primed identifiers, identifiers followed by the prime symbol ''', are used to denote next-state variables.

2.3.1 Definition (Identifier)**.** The set **Id** of identifiers is defined inductively as follows:

- Any name is an identifier.
- If id_1 and id_2 are identifiers then $\mathrm{id}_1\ .\ \mathrm{id}_2$ is also an identifier.

Priming an identifier adds the prime symbol to the identifer. The set of primed identifiers is denoted by $\mathbf{Id}'$. If $id \in \mathbf{Id}$ then $id' \in \mathbf{Id}'$ denotes its primed version. If V is a set of identifiers, V' denotes the set $\{\ id' \mid id \in V\ \}$.

Let $\mathbf{B}$ denote the set $\{\ 0,\ 1\ \}$. Bitvector constants are finite words over the alphabet $\mathbf{B}$.

2.3.2 Definition (Bitvector Constant)**.** For $n \geq 1$, let $\mathbf{B}^n$ denote the set of bitvector constants of size n. The set of all bitvectors $\bigcup_{n \geq 1} \mathbf{B}^n$ is denoted by $\mathbf{B}^+$. Given a bitvector constant $w \in \mathbf{B}^n$ and $i \leq n$, $w(i)$ denotes the i-th bit of w.

Variables are identifiers with a type. The type of a variable is the set $\mathbf{B}^n$, for some $n \geq 1$.

2.3.3 Definition (Variables)**.** $V \subseteq \mathbf{Id}$ is called a set of variables if it is associated with a type function *Ty*

$$Ty : V \to \{\, \mathbf{B}^n \,\}_{n \geq 1}$$

*Ty*is extended over $V' \subseteq \mathbf{Id}'$: $Ty(v') = Ty(v)$. Primed variables refer to next-state variables. If v is a current state variable, v' denotes its next-state version. Similarly, if $v' \in V'$ is a next-state variable, v denotes its current-state version.

2.3.4 Definition (Environment)**.** An environment over the set of variables V is a function

$$e : V \to \mathbf{B}^+$$

that assigns each variable a value of its type:

$$\forall\, v \in V.\ e(v) \in Ty(v)$$

If V_1 and V_2 are disjoint sets of variables and e_1 and e_2 are environments defined over V_1 and respectively, V_2, their union $e_1 \cup e_2$ is the environment over $V_1 \cup V_2$ defined by

$$(e_1 \cup e_2)(v) = \begin{cases} e_1(v) & : v \in V_1 \\ e_2(v) & : v \in V_2 \end{cases}$$

Let $V \subseteq \mathbf{Id}$ and let e be an environment over V. The environment e' over V' is defined by $e'(v') = e(v)$.

Consider a set of variables V_1. We say V_2 is a copy of V_1 if there exists a bijective function $\phi : V_1 \to V_2$ such that

$$\forall\, v \in V_1.\ Ty(v) = Ty(\phi(v))$$

If V_2 is a copy of V_1 and $e_1 \in Env(V_1)$ we denote by $e_1\left[{}^{V_2}/_{V_1}\right]$ the environment $e_2 \in Env(V_2)$ such that:

$$\forall\, v \in V_2.\ e_2(v) = e_1(\phi^{-1}(v))$$

2.3.5 Definition (Bitvector Expression)**.** Let $V \subseteq \mathbf{Id}$ be a set of variables and *Ty* be the associated type function. Bitvectors are typed expressions over constants in $\mathbf{B}^+$ and variables in V. We say that the bitvector t has type $\mathbf{B}^n$ using the typing expression $t : \mathbf{B}^n$. If $t : \mathbf{B}$ we say t is single-bit.

Base Case

- If $v \in V$ is a variable such that $Ty(v) = \mathbf{B}^n$, then v is a bitvector expression of type $\mathbf{B}^n$.
- If $w \in \mathbf{B}^n$ is a constant, then w is a bitvector of type $\mathbf{B}^n$.

Inductive Case Bitvector expressions are constructed using a set of operators. For such an operator $\mathbf{op}$, a type rule is used to denote the type requirements on its operands and the type of its application:

$$\frac{t_1 : \mathbf{B}^{n_1}, \ldots, t_k : \mathbf{B}^{n_k}}{\mathbf{op}(t_1, \ldots, t_k) : \mathbf{B}^n}$$

If t is a bitvector expression then *vars*(t) stands for the set of variables that appear syntactically in t. Similarly, *consts*(t) denotes the set of constants $w \in \mathbf{B}^+$ that appear syntactically in t.

Variables are assigned values by an environment e over V. The semantics of a constant $w : \mathbf{B}^n$ is $[\![w]\!]_e = w$. The semantics of a variable $v : \mathbf{B}^n$ is $[\![v]\!]_e = e(v)$. The semantics of a bitvector expression t such that $t : \mathbf{B}^n$ is denoted by $[\![t]\!]_e \in \mathbf{B}^n$.

Subexpression

$$\begin{array}{ll} \text{Type Rule} & \dfrac{t : \mathbf{B}^n}{(t) : \mathbf{B}^n} \\ \text{Semantics} & [\![(t)]\!]_e = [\![t]\!]_e \end{array}$$

Constant

$$\begin{array}{ll} \text{Type Rule} & \dfrac{w \in \mathbf{B}^n}{w : \mathbf{B}^n} \\ \text{Semantics} & [\![w]\!]_e = w \end{array}$$

Variable

$$\begin{array}{ll} \text{Type Rule} & \dfrac{v \in V, \quad \mathit{Ty}(v) = \mathbf{B}^n}{v : \mathbf{B}^n} \\ \text{Semantics} & [\![v]\!]_e = e(v) \end{array}$$

Bitwise Operators The bitvector operator $\mathbf{op} \in \{\, \mathbf{not}, \mathbf{and}, \mathbf{or}, \mathbf{xor}, \ldots \}$ is defined using the corresponding Boolean operator $\mathbf{op}_2 \in \{\, \mathbf{not}_2, \mathbf{and}_2, \mathbf{or}_2, \mathbf{xor}_2, \ldots \}$.

$$\begin{array}{ll} \text{Type Rule} & \dfrac{t_1 : \mathbf{B}^n,\ t_2 : \mathbf{B}^n}{t_1\ \mathbf{op}\ t_2 : \mathbf{B}^n} \\ \text{Semantics} & [\![t_1\ \mathbf{op}\ t_2]\!]_e\,(i) = [\![t_1]\!]_e\,(i)\ \mathbf{op}_2\ [\![t_2]\!]_e\,(i),\ i = \overline{0,\ n-1} \end{array}$$

Bitwise negation requires only one argument and we present it separately.

$$\begin{array}{ll} \text{Type Rule} & \dfrac{t : \mathbf{B}^n}{\mathbf{not}\ t : \mathbf{B}^n} \\ \text{Semantics} & [\![\mathbf{not}\ t]\!]_e\,(i) = \mathbf{not}_2 \left(\ [\![t]\!]_e\,(i) \ \right),\ i = \overline{0,\ n-1} \end{array}$$

Subrange Let $0 \leq l \leq m$. The subrange operator $[l : m]$ extracts the sequence of bits l, $l+1$, $\ldots$ m.

$$\begin{array}{ll} \text{Type Rule} & \dfrac{t : \mathbf{B}^n,\ m \leq n}{t[l : m] : \mathbf{B}^{m-l+1}} \\ \text{Semantics} & [\![t[l : m]]\!]_e\,(i) = [\![t]\!]_e\,(i + l),\ i = \overline{0,\ m-l} \end{array}$$

Concatenation The concatenation operator :: appends the second argument to the first one.

$$\text{Type Rule} \quad \frac{t_1 : \mathbf{B}^{n_1},\, t_2 : \mathbf{B}^{n_2}}{t_1 :: t_2 : \mathbf{B}^{n_1+n_2}}$$

$$\text{Semantics} \quad [\![t_1 :: t_2]\!]_e\,(i) = \begin{cases} [\![t_1]\!]_e\,(i) & : \quad i < n_1 \\ [\![t_2]\!]_e\,(i - n_1) & : \quad i \geq n_1 \end{cases},\ i = \overline{0,\, n_1 + n_2 - 1}$$

Equality The equality operator $=$ compares its operands for equality.

$$\text{Type Rule} \quad \frac{t_1 : \mathbf{B}^{n},\, t_2 : \mathbf{B}^{n}}{t_1 = t_2 : \mathbf{B}}$$

$$\text{Semantics} \quad [\![t_1 = t_2]\!]_e = \begin{cases} 0 & : \quad \text{if } [\![t_1]\!]_e \text{ not equals } [\![t_2]\!]_e \\ 1 & : \quad \text{if } [\![t_1]\!]_e \text{ equals } [\![t_2]\!]_e \end{cases}$$

Assignment The assignment operator $:=$ is a special case of equality when the left operand is a variable.

$$\text{Type Rule} \quad \frac{v : \mathbf{B}^{n},\, t : \mathbf{B}^{n}}{v := t : \mathbf{B}}$$

$$\text{Semantics} \quad [\![v := t]\!]_e = \begin{cases} 0 & : \quad \text{if } e(v) \text{ not equals } [\![t]\!]_e \\ 1 & : \quad \text{if } e(v) \text{ equals } [\![t]\!]_e \end{cases}$$

Nondeterministic Assignment It is a special case of assignment when the right hand side of the assignment is the expression $\mathbf{choice}$.

$$\text{Type Rule} \quad \frac{v : \mathbf{B}^{n}}{v := \mathbf{choice} : \mathbf{B}}$$

$$\text{Semantics} \quad [\![v := \mathbf{choice}]\!]_e = 1$$

If-then-else The operator *if-then-else* interprets its first operand as a Boolean and selects between the second and third arguments accordingly.

$$\text{Type Rule} \quad \frac{t_1 : \mathbf{B},\, t_2 : \mathbf{B}^{n},\, t_3 : \mathbf{B}^{n}}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : \mathbf{B}^{n}}$$

$$\text{Semantics} \quad [\![\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3]\!]_e = \begin{cases} [\![t_2]\!]_e & : \quad \text{if } [\![t_1]\!]_e \text{ equals } 1 \\ [\![t_3]\!]_e & : \quad \text{if } [\![t_1]\!]_e \text{ equals } 0 \end{cases}$$

Arithmetic Operators We allow bitvector expressions using the standard arithmetic operators $\mathbf{op}$ in the set $\{+, -, *, \ldots\}$. The semantics of arithmetic operations uses two functions that convert bitvectors to integers and conversely, integers to bitvectors: $\mathit{integer} : \mathbf{B}^{+} \to \mathbb{Z}$ and $\mathit{repr} : \mathbb{Z} \times \mathbb{N} \to \mathbf{B}^{+}$ such that if $w : \mathbf{B}^{n}$ then

$$\mathit{repr}(\mathit{integer}(w),\, n) = w$$

The exact definition of *integer* and *repr* depends on whether bitvector arguments are interpreted as signed or unsigned. For the signed case, two's complement is a suitable representation. For unsigned operations we use the binary representation of positive integers. The choice of these encodings and the semantics of overflowing operations does not influence upon the remaining work.

$$\text{Type Rule} \qquad \frac{t_1 : \mathbf{B}^n,\ t_2 : \mathbf{B}^n}{t_1\ \mathbf{op}\ t_2 : \mathbf{B}^n}$$

$$\text{Semantics} \quad [\![t_1\ \mathbf{op}\ t_2]\!]_e = \mathit{repr}\Big(\ \mathit{integer}([\![t_1]\!]_e)\ \mathbf{op}\ \mathit{integer}([\![t_2]\!]_e),\ n\ \Big)$$

Relational Operators Bitvectors can be compared using the integer relational operators $\mathbf{op}$ in the set $\{ <, \leq, \geq, > \}$.

$$\text{Type Rule} \qquad \frac{t_1 : \mathbf{B}^n,\ t_2 : \mathbf{B}^n}{t_1\ \mathbf{op}\ t_2 : \mathbf{B}^n}$$

$$\text{Semantics} \quad [\![t_1\ \mathbf{op}\ t_2]\!]_e = \begin{cases} 0 & : \quad \text{if } \mathit{integer}([\![t_1]\!]_e)\ \mathbf{op}\ \mathit{integer}([\![t_2]\!]_e) \\ 1 & : \quad \text{if not } \mathit{integer}([\![t_1]\!]_e)\ \mathbf{op}\ \mathit{integer}([\![t_2]\!]_e) \end{cases}$$

We fix the precedence of operators from high to low to be the following one:

not
$[:]$
::
$*$
$+ \quad -$
$< \quad \leq \quad = \quad \geq >$
and
or **xor**
if-then-else
$:=$

2.3.6 Example. Consider $V = \{ v_1, v_2, v_3, v_4 \}$ and $e \in \mathit{Env}(V)$ defined as follows:

variable	*Ty*	e
v_1	$\mathbf{B}^1$	*1*
v_2	$\mathbf{B}^2$	*10*
v_3	$\mathbf{B}^3$	*011*
v_4	$\mathbf{B}^4$	*1011*

Examples of bitvector expression and their semantics is given below:

expression	$[\![\cdot]\!]_e$
v_4	*1011*
$v_1 :: v_3$	*1011*
v_4 **xor** $(v_1 :: v_3)$	*0000*
$\mathbf{not} v_4$	*0100*
$(v_1 :: v_3)[1:3]$	*011*
$(v_1 :: v_3)[1:3] = v_3$	*1*
$v_3 :=$ *111*	*0*
$\mathbf{not}(v_1 :: v_3)[1:3] = v_3)$	*0*
$\mathbf{if\ not}((v_1 :: v_3)[1:3] = v_3)\ \mathbf{then}\ v_4\ \mathbf{else\ not} v_4$	*0100*
$v_2 +$ *01*	*11*
$v_4 \leq$ *1100*	*1*

Formulas over V are defined inductively: basic formulas are single-bit bitvectors and complex formulas are formed from simple ones using Boolean connectives. Semantically formulas identify with a subset of the single-bit bitvectors.

2.3.7 Definition (Formula)**.**

Syntax

- If $t : \mathbf{B}$ is a single-bit bitvector then t is a formula.
- If f, f_1, f_2 are formulas then $\neg f$, $f_1 \wedge f_2$, $f_1 \vee f_2$ are formulas.

Semantics The truth value of a formula f under a given environment e that valuates its variables is defined by induction over its structure. If f is true in e, we write $e \models f$. We define $\models$ as follows:

- If f is the Boolean bitvector t then

$$e \models f \text{ if } [\![t]\!]_e = 1$$

- If f has form $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ then

$$e \models \neg f_1 \text{ if } e \not\models f_1$$
$$e \models f_1 \wedge f_2 \text{ if } \models f_1 \text{ and } e \models f_2$$
$$e \models f_1 \vee f_2 \text{ if } e \models f_1 \text{ or } e \models f_2$$

2.3.8 Definition (Circuit)**.** A circuit is a tuple

$$C = \langle V_r, V_c, \mathit{Ty}, V_i, V_o, \mathit{Insts}, \mathit{Init}, \mathit{Tr} \rangle$$

- V_r and V_c are disjoint sets of identifiers. V_r denotes the set of register variables (also called state variables or current-state variables). V_c denotes the combinational variables. $V_r{}'$ denotes the next-state variables.
- $\mathit{Ty} : V_r \cup V_c \to \{ \mathbf{B}^n \}_{n \geq 1}$ is the type function.
- V_i and V_o are disjoint subsets of $V_r \cup V_c$ and represent the input and respectively, output variables.
- Input variables are required to be combinational: $V_i \subseteq V_c$.
- *Insts* is a set of circuit instances.
- *Init* is a set of assignments to current state variables.
- *Tr* is a set of assignments to combinational and next-state variables and contains exactly one assignment of form $v := t$, where t is defined over variables in $V_r \cup V_c$, for each $v \in V_r{}' \cup V_c$.

Circuits are defined inductively:

Base case $\mathit{Insts} = \emptyset$

Inductive case $\mathit{Insts} = \{ inst_1, \ldots, inst_n \}$

2.3.9 Definition (Circuit Instance)**.** A circuit instance is a tuple $inst = \langle \mathit{id}, C, \mathit{InputArg}, \mathit{OutputArg} \rangle$ where:

- *id* is an identifier
- C is a circuit
- *InputArg* is a set of bitvector expressions in bijection with $C.V_i$ and denotes the input arguments
- *OutputArg* is a set of combinational or next-state variables of the enclosing circuit, in bijection with $C.V_o$ and denotes the output arguments

The instance $inst$ stands for a copy $\phi_{inst.id}(C)$ of C with variables and instance names renamed by the mapping $\phi_{inst.id} : \mathbf{Id} \cup \mathbf{Id}' \to \mathbf{Id} \cup \mathbf{Id}'$ defined by

$$\phi_{inst.id}(id) = id \ . \ id$$

The mapping $\phi_{inst.id}$ naturally extends to bitvector expressions, to formulas, to sets of such objects and to functions over such sets. Given a bitvector t, $\phi_{inst.id}(t)$ is obtained by substituting each occurrence of a variable v in t by $\phi_{inst.id}(v)$. Renaming preserves the variable types:

$$\phi_{inst.id}(\mathit{Ty})(\phi_{inst.id}(v)) = \mathit{Ty}(v)$$

Instances $inst_i = \langle id_i, C_i, InputArg_i, OutputArg_i \rangle$ of C are renamed to

$$\phi_{inst.id}(inst_i) = \langle \phi_{inst.id}(id_i), C_i, \phi_{inst.id}(InputArg_i), \phi_{inst.id}(OutputArgv_i) \rangle$$

Let $inst$ be an instance of a circuit C. Let $v \in inst.C.V_i \cup inst.C.V_o$. We define $\mathit{Arg}(C, inst.v)$ to be the actual argument to $inst.v$ in C. When the context is clear, we only write $\mathit{Arg}(v)$.

Let C be a circuit and let $C.\mathit{Insts} = \{ inst_1, \ldots, inst_n \}$. We use the notation

$$\mathit{Labels}(C.\mathit{Insts}) = \{ inst_1.\mathit{id}, \ldots, inst_n.\mathit{id} \}$$

If $id \in \mathit{Labels}\ C.\mathit{Insts}$ then $C.\mathit{Insts}[id]$ denotes $inst \in C.\mathit{Insts}$ such that $inst.\mathit{id} = id$.

In the next example we introduce syntactic sugar to represent circuits textually.

2.3.10 Example. In Figure 2.4 we describe a 1-bit adder implemented using two half-adders. Lines 1–5 describe the HalfAdder circuit:

$$\begin{aligned}
V_r &= \emptyset \\
V_c &= \{ a, b, sum, cout \} \\
V_i &= \{ a, b \} \\
V_o &= \{ sum, cout \} \\
\mathit{Insts} &= \emptyset \\
\mathit{Init} &= \emptyset \\
\mathit{Tr} &= \{ sum := a \ \mathbf{xor}\ b, cout := a \ \mathbf{and}\ b \}
\end{aligned}$$

Lines 6–14 describe the Adder circuit:

$$V_r = \emptyset$$

```
1  ckt HalfAdder(a, b : bitvec[1])(sum, cout : bitvec[1])
2    assign
3      sum := a xor b;
4      cout := a and b;
5  end

7  ckt Adder (a, b, cin : bitvec[1])(sum, cout : bitvec[1])
8    var
9      sum1, cout1, cout2 : bitvec[1];
10   inst
11     ha1 : HalfAdder(a, b)(sum1, cout1)
12     ha2 : HalfAdder(sum1, cin)(sum, cout2)
13   assign
14     cout := cout1 or cout2;
15 end
```

Figure 2.4: Adder circuit.

$$
\begin{aligned}
V_c &= \{\, a, b, cin, sum, cout, sum_1, cout_1, cout_2 \,\} \\
V_i &= \{\, a, b, cin \,\} \\
V_o &= \{\, sum, cout \,\} \\
\mathit{Insts} &= \{\, \langle ha_1, \text{HalfAdder}, \{\, a, b \,\}, \{\, sum_1, cout_1 \,\}\rangle, \\
&\qquad \langle ha_2, \text{HalfAdder}, \{\, sum_1, cin \,\}, \{\, sum, cout_2 \,\}\rangle\} \\
\mathit{Init} &= \emptyset \\
\mathit{Tr} &= \{\, cout := cout_1 \textbf{ or } cout_2 \,\}
\end{aligned}
$$

2.3.11 Definition (Variable Dependency Graph)**.** Let $C = \langle V_r, V_c, \mathit{Ty}, V_i, V_o, \mathit{Insts}, \mathit{Init}, \mathit{Tr}\rangle$. The variable dependency graph is a digraph $\mathit{VarDepGraph}(C) = \langle \mathit{Nodes}, \mathit{Succ}\rangle$:

$$
\begin{aligned}
\mathit{Nodes} &\equiv V_r \cup V_c \cup V_r{}' \\
\mathit{Succ} &\equiv \{\, (v_1, v_2) \mid \exists\, t \in \mathit{Expr}(V_r \cup V_c \cup \mathbf{B}^+).\ v_2 := t \in \mathit{Tr} \wedge v_1 \in \mathit{vars}(t) \,\} \cup \mathit{Succ}_{insts}
\end{aligned}
$$

The set Succ_{insts} is defined inductively. We write $v_1 \preceq_{var} v_2$ to denote that v_1 and v_2 are in the transitive closure of *Succ*.

Base Case If $\mathit{Insts} = \emptyset$ then $\mathit{Succ}_{insts} \equiv \emptyset$.

Inductive Case

$$\begin{aligned}\mathit{Succ}_{insts} \equiv \bigcup_{inst \in \mathit{Insts}} \{\, (v_1, v_2) \mid\ & \exists\, v_i \in inst.C.V_i. \\ & \exists\, v_o \in inst.C.V_o. \\ & \exists\, t \in \mathit{Expr}(V_r \cup V_c \cup \mathbf{B}^+). \\ & \quad v_i \preceq_{var} v_o \wedge \\ & \quad t = \mathit{Arg}(v_i) \wedge \\ & \quad v_2 = \mathit{Arg}(v_o) \wedge \\ & \quad v_1 \in \mathit{vars}(t) \,\}\end{aligned}$$

In order to simplify the analysis of circuits we require that all circuits have cycle free variable dependency graphs.

2.3.12 Definition (Circuit Elaboration)**.** Circuit elaboration gives a precise semantics to circuit instantiation. The elaboration of a circuit C, denoted by by $\mathit{Elab}(C)$, is a circuit that has no instances:

$$\mathit{Elab}(C).\mathit{Insts} = \emptyset$$

$\mathit{Elab}(C)$ is defined by induction over the structure of C.

- If $C.\mathit{Insts} = \emptyset$ then
$$\mathit{Elab}(C) = C$$
- If $C.\mathit{Insts} = \{\, \langle id_1, C_1, InputArg_1, OutputArg_1\rangle, \ldots, (\langle id_n, C_n, InputArg_n, OutputArg_n\rangle \,\}$ then

$$\begin{aligned}
\mathit{Elab}(C).V_r &= C.V_r \cup \bigcup_{i=1}^{n} \Big(\ \mathit{Elab}(\phi_{\mathrm{id}_i}(C_i))\ \Big).V_r \\
\mathit{Elab}(C).V_c &= C.V_c \cup \bigcup_{i=1}^{n} \Big(\ \mathit{Elab}(\phi_{\mathrm{id}_i}(C_i))\ \Big).V_c \\
\mathit{Elab}(C).\mathit{Ty} &= C.\mathit{Ty} \cup \bigcup_{i=1}^{n} \Big(\ \mathit{Elab}(\phi_{\mathrm{id}_i}(C_i))\ \Big).\mathit{Ty} \\
\mathit{Elab}(C).V_i &= C.V_i \\
\mathit{Elab}(C).V_o &= C.V_o \\
\mathit{Elab}(C).\mathit{Init} &= C.\mathit{Init} \cup \bigcup_{i=1}^{n} \Big(\ \mathit{Elab}(\phi_{\mathrm{id}_i}(C_i))\ \Big).\mathit{Init} \\
\mathit{Elab}(C).\mathit{Tr} &= C.\mathit{Tr} \cup \bigcup_{i=1}^{n} \Big(\ \mathit{Elab}(\phi_{\mathrm{id}_i}(C_i))\ \Big).\mathit{Tr} \cup \mathit{InputAsns} \cup \mathit{OutputAsns}
\end{aligned}$$

where

$$InputAsns \equiv \bigcup_{i=1}^{n} \left\{ \; id_i.v := \mathit{InputArg}(inst_i.v) \mid v \in \left(\; \mathit{Elab}(\phi_{\mathrm{id}_i}(C_i)) \; \right).V_i \; \right\} \quad (2.6)$$

$$OutputAsns \equiv \bigcup_{i=1}^{n} \left\{ \; \mathit{OutputArg}(inst_i.v) := id_i.v \mid v \in \left(\; \mathit{Elab}(\phi_{\mathrm{id}_i}(C_i)) \; \right).V_i \; \right\}$$

Equation 2.6 explains why input variables and the arguments of output variables must be combinational: the semantics of input and output arguments to circuit intances is given by variable assigment, and variable assignment is allowed only to combinational or next-state variables.

2.3.13 Example. We elaborate the adder circuit introduced in Example 2.3.10. The adder has two instances ha_1 and ha_2 of the HalfAdder circuit. The two instances are renamed. By applying ϕ_{ha_1} to HalfAdder we get:

$$\begin{aligned}
\phi_{ha_1}(\text{HalfAdder}).V_r &= \emptyset \\
\phi_{ha_1}(\text{HalfAdder}).V_c &= \{\, ha_1.a, ha_1.b, ha_1.sum, ha_1.cout \,\} \\
\phi_{ha_1}(\text{HalfAdder}).V_i &= \{\, ha_1.a, ha_1.b \,\} \\
\phi_{ha_1}(\text{HalfAdder}).V_o &= \{\, ha_1.sum, ha_1.cout \,\} \\
\phi_{ha_1}(\text{HalfAdder}).\mathit{Insts} &= \emptyset \\
\phi_{ha_1}(\text{HalfAdder}).\mathit{Init} &= \emptyset \\
\phi_{ha_1}(\text{HalfAdder}).\mathit{Tr} &= \{\, ha_1.sum := ha_1.a \;\mathbf{xor}\; ha_1.b, ha_1.cout := ha_1.a \;\mathbf{and}\; ha_1.b \,\}
\end{aligned}$$

Similarly, applying ϕ_{ha_2} to HalfAdder yields:

$$\begin{aligned}
\phi_{ha_2}(\text{HalfAdder}).V_r &= \emptyset \\
\phi_{ha_2}(\text{HalfAdder}).V_c &= \{\, ha_2.a, ha_2.b, ha_2.sum, ha_2.cout \,\} \\
\phi_{ha_2}(\text{HalfAdder}).V_i &= \{\, ha_2.a, ha_2.b \,\} \\
\phi_{ha_2}(\text{HalfAdder}).V_o &= \{\, ha_2.sum, ha_2.cout \,\} \\
\phi_{ha_2}(\text{HalfAdder}).\mathit{Insts} &= \emptyset \\
\phi_{ha_2}(\text{HalfAdder}).\mathit{Init} &= \emptyset \\
\phi_{ha_2}(\text{HalfAdder}).\mathit{Tr} &= \{\, ha_2.sum := ha_2.a \;\mathbf{xor}\; ha_2.b, ha_2.cout := ha_2.a \;\mathbf{and}\; ha_2.b \,\}
\end{aligned}$$

Since $\phi_{ha_1}(\text{HalfAdder}).\mathit{Insts} = \emptyset$ and $\phi_{ha_2}(\text{HalfAdder}).\mathit{Insts} = \emptyset$ we have

$$\begin{aligned}
\mathit{Elab}(\phi_{ha_1}(\text{HalfAdder})) &= \phi_{ha_1}(\text{HalfAdder}) \\
\mathit{Elab}(\phi_{ha_2}(\text{HalfAdder})) &= \phi_{ha_2}(\text{HalfAdder})
\end{aligned}$$

We proceed to the elaboration of the Adder:

$$
\begin{aligned}
\mathit{Elab}(\text{Adder}).V_r &= \emptyset \\
\mathit{Elab}(\text{Adder}).V_c &= \{\, a, b, cin, sum, cout, sum_1, cout_1, cout_2 \,\} \cup \\
&\quad \{\, ha_1.a, ha_1.b, ha_1.sum, ha_1.cout \,\} \cup \\
&\quad \{\, ha_2.a, ha_2.b, ha_2.sum, ha_2.cout \,\} \\
\mathit{Elab}(\text{Adder}).V_i &= \{\, a, b, cin \,\} \\
\mathit{Elab}(\text{Adder}).V_o &= \{\, sum, cout \,\} \\
\mathit{Elab}(\text{Adder}).\mathit{Insts} &= \emptyset \\
\mathit{Elab}(\text{Adder}).\mathit{Init} &= \emptyset \\
\mathit{Elab}(\text{Adder}).\mathit{Tr} &= \{\, cout := cout_1 \ \mathbf{or}\ cout_2 \,\} \cup \\
&\quad \{\, ha_1.sum := ha_1.a \ \mathbf{xor}\ ha_1.b, ha_1.cout := ha_1.a \ \mathbf{and}\ ha_1.b \,\} \cup \\
&\quad \{\, ha_2.sum := ha_2.a \ \mathbf{xor}\ ha_2.b, ha_2.cout := ha_2.a \ \mathbf{and}\ ha_2.b \,\}
\end{aligned}
$$

Given a set of variable assignments Asn we define the formula associated with it by

$$\mathit{formula}(Asn) \equiv \bigwedge_{v:=t \in Asn} v := t$$

2.3.14 Definition (Circuit Semantics)**.** The semantics of a circuit C is given as a labeled transition system

$$\mathit{LTS}(C) = \langle Q_C, R_C, T_C, I_C \rangle$$

- The set of states Q_C is the set of environments over $\mathit{Elab}(C).V_r$.
- The set of transition labels T_C is the set of environments over $\mathit{Elab}(C).V_c$.
- The transition relation $R_C \subseteq Q_C \times T_C \times Q_C$ is defined by the assignments $\mathit{Elab}(C).\mathit{Tr}$:

$$(q_C, t_C, {q_C}') \in R_C \equiv (q_C \cup t_C \cup {q_C}' \left[{V_r}' / V_r \right]) \models \mathit{formula}(\mathit{Elab}(C).\mathit{Tr})$$

- The set of intial states is defined by the assignments $\mathit{Elab}(C).\mathit{Init}$:

$$q_C \in I_C \equiv q_C \models \mathit{formula}(\mathit{Elab}(C).\mathit{Init})$$

2.3.15 Example. Figure 2.5 and Figure 2.6 describe two two-bit counters. Each figure shows the circuit description and the corresponding labeled transition system defined in Definition 2.3.14. Both circuits have a register variable for the counter value; they differ with respect to the increment.

```
1  ckt counter(inc : bool)(count : bitvec[2])
2    assign
3      count′ := if inc then count + 1 else count;
4  end
```

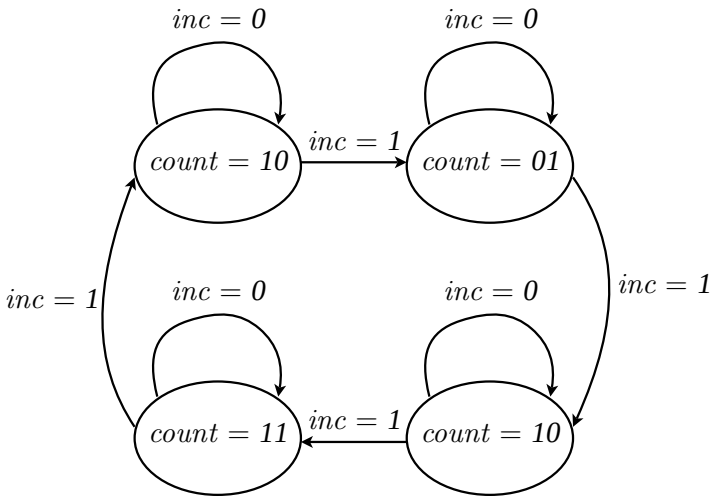


Figure 2.5: Counter With Combinational Increment

In one the increment is a combinational input and in the other it is a register. The two counters have the same language with respect to the value of the counter variable. However they are not bisimilar: the counter with combinational increment simulates the one with registered increment, but not viceversa.

2.3.16 Proposition. Let C be a circuit such that all its instances are combinational. Let $q_C \in Q_C$, $q_C{}' \in Q_C$ and $t_C \in \mathit{Env}(V_c)$. If the following conditions hold:

$$\forall \text{`}v := expr\text{'} \in \mathit{Tr}.\ (t_C \cup q_C{}' \left[{V_r}' / V_r \right])(v) = [\![expr]\!]_{q_C \cup q_C} \tag{2.7}$$

$$\begin{aligned} &\forall\ inst \in \mathit{Insts}.\ \exists\ (q_I, t_I, q_I{}') \in \mathit{LTS}(inst.C).R_C. \\ &\quad \forall\ v \in inst.C.V_i \cup inst.C.V_o.\ t_I(v) = t_C(\mathit{Arg}(v)) \end{aligned} \tag{2.8}$$

then there exists $t_{C1} \in T_C$ such that $t_C \subseteq t_{C1}$ and $(q_C, t_{C1}, q_C{}')$ is a step of $\mathit{LTS}(C)$, i.e. $(q_C, t_{C1}, t_C{}') \in R_C$.

```
6   ckt counter(inci : bool)(count : bitvec[2])
7     var
8       inc : bool;
9     assign
10      count' := if inc then count + 1 else count;
11      inc' := inci;
12  end
```

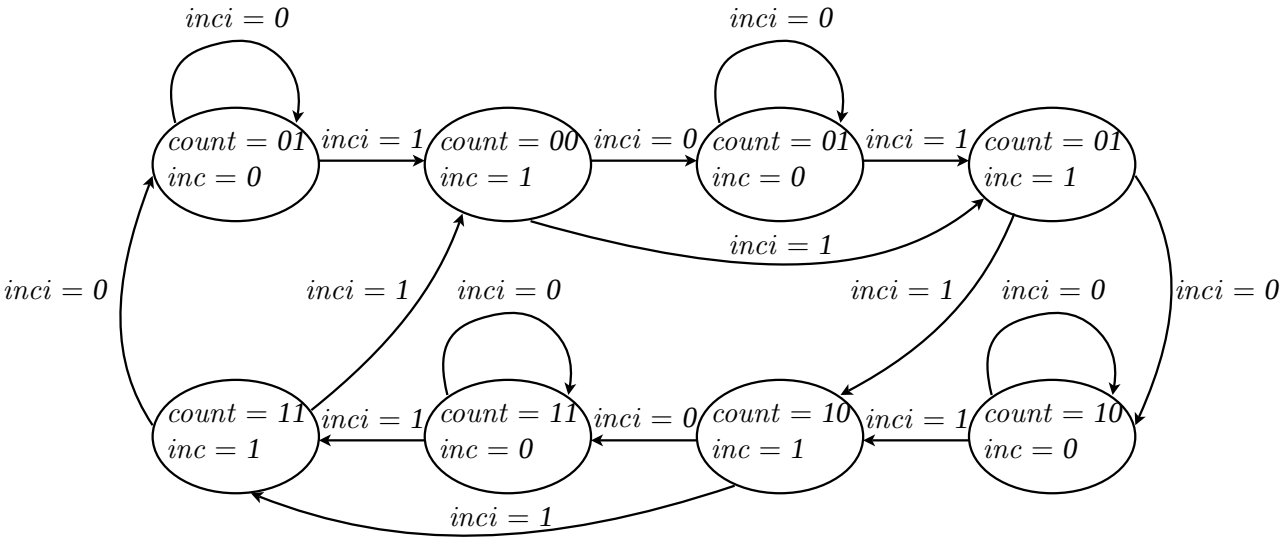


Figure 2.6: Counter With Registered Increment

Proof. We define t_{C1} as follows:

$$\forall\, inst_i \in \mathit{Insts}.\ \forall\, v \in (\mathit{Elab}(inst_i.C)).V_c.$$
$$t_{C1}(id_i.v) = t_{Ii}(v)$$

It follows that $q_{Pc} \cup t_{C1} \cup q_{Pc}{}' \left[{V_r}' / V_r \right])$ satisfies all the assignments in $\mathit{Elab}(C)$ since the restriction of t_{C1} to the variables of each elaborated instance equals the transition label of the instance.

□

2.4 Related Work

Pipelining is a prevalent technique to improve the throughput of hardware circuits. The most advanced use of pipelining occurs in microprocessors circuits. Formal verification and other validation techniques that target microprocessors will therefore deal implicitly or explicitly with the challenges of pipelining. We catalog the well known approaches to microprocessor verification in Section 2.4.1. We describe pipeline specific approaches in Section 2.4.2. The current body of work on datapath abstraction is summarized in Section 2.4.3.

2.4.1 Microprocessor Verification

The challenges of microprocessor verfication manifest at two main levels: specification and the state explosion problem. First, a verfication technique must formulate correctness of the implementation with respect to a reference specification. Second, such a methodology must address the state explosion problem by casting the correctness problem in a logic with an efficient decision procedure or decomposition and abstraction techniques.

In the case of microprocessors, the specification is readily available in the form of the instruction set architecture. At this level, the specification is often thought of as a circuit that executes one instruction at a time. Each type of instruction is described in terms of the side effects it performs on the specification state which consists of the programmer visible state holding elements such as the program counter, register file and memory. A pipelined microprocessor overlaps the execution of several instructions at a time and updates to its state holding elements are performed by multiple instructions in various stages of execution.

One of the well accepted correctness criteria is Milner's simulation [Milner, 1971]. The challenge in applying simulation as a correctness statement for microprocessors is to align the specification state with the implementation state. Microprocessor designs usually allow for an execution mode called flushing that continues the execution of inflight instructions but prevents new ones from being fetched and entering the pipeline. Burch and Dill [1994] were the first to use flushing to formulate a correctness criterion. Their method uses an abstraction function that flushes the implementation state and then projects out the programmer visibile state: the program counter, register file and memory. The abstraction function reduces to projectio when the pipeline is in the beginning and end state of a computation. Proving the commuting diagram in Milner's simulation implies correctness with respect to the programmer visible state.

Burch and Dill model the pipeline datapath using uninterpreted functions and reduce the verification of the commuting diagram to a decision problem in a restricted logic with uninterpreted functions and positive equality. The scalability of the approach is limited by the capacity of the decision

procedure and therefore by the size of the terms that describe the commuting diagram. The terms that describe flushing increase proportionally to the number of cycles needed to flush the implementation. For simple linear pipelines, this corresponds to the number of inflight instructions. For complex pipelines with out-of-order execution and variable instruction latencies, flushing becomes more expensive.

Velev and Bryant [Velev and Bryant, 2000, Velev, 2001] extend the capacity of the Burch-Dill flushing technique with customized rewrite rules that target the various types of logic subterms encountered in flushing the implementation. They also migrate the underlying decision procedure to a SAT based implementation [Velev and Bryant, 2003]. A similar approach to the verification of an XScale microprocessor has been applied in Srinivasan and Velev [2003].

Another way to improve flushing is through compositional verification [Levitt and Olukotun, 1997, Skakkebak et al., 1998, Hosabettu et al., 1998]. The method of completion functions of Hosabettu et al. decomposes the commuting diagram based on flushing into several commuting diagrams each dealing with a particular stage of the pipeline. The proof of the diagram for a particular stage assumes the correctness of the downstream stages. For a linear pipeline of length n, there are n commuting diagrams. For more complex pipelines, the method suffers from a combinatorial explosion in the number of possible paths through the pipeline. Initially proposed in a theorem prover setting, the modularity of completion functions was leveraged using an off-the-shelf equivalence checker on several RTL design [Aagaard et al., 2004]. Equivalence checking techniques were also used by Appenzeller and Kuehlmann [1995] to verify a PowerPC microprocessor and by Bhagwati and Devadas [1994] to verify a reduced Alpha design.

The logic of positive equality of Burch and Dill was extended in the UCLID verifier [Lahiri et al., 2002]. The decision procedure was implemented by translation to SAT. The translation does however suffer from false negatives and requires manual effort to debug the counterexamples. Proofs done in UCLID are inductive and in most cases benefit from user generated invariants. Other approaches to microprocessor verification that use UCLID include that of Manolios and Srinivasan [2004], Andraus and Sakallah [2004].

General theories for conducting microprocessor correctness proofs in a theorem proving setting have been presented [Windley, 1995, Huggins and Campenhout, 1998]. Miller and Srivas [1995] describe using the PVS theorem prover to prove a commuting diagram based correctness statement for an industrial CISC microprocessor. The effort put in the complete verification was over 3000 hours. Sawada and Hunt [1997] use the ACL2 theorem prover [Kaufmann and Moore, 1997] to verify an out-of-order microprocessor with dynamic resolution of data hazards. The toplevel correctness is stated using flushing. The decomposition of the proof is based enhancing the execution trace of the microprocessor with a table of history variables called MAETT. The MAETT facilitates the proof of invariants.

There are also approaches that use a mix of theorem proving for proof decomposition and model checking for the verification of the control. McMillan [1998] extends the SMV model checker [McMillan, 1992] with support for compositional reasoning and assume-guarantee style proofs. His approach leverages symmetric data types [Norris IP and Dill, 1996]. A mix of theorem proving and model checking with uninterpreted functions is also used by Berezin et al. [1998], Jacobi [2002].

2.4.2 Hazard-Based Verification Techniques

Validation methods that use pipeline hazards fall into several categories: property specification and correctness statements, decomposition and abstraction, and automatic test pattern generation.

Aagaard [2003] describes a correctness statement that refers at the toplevel only to the three types of hazard correctness: datapath, control and structural. Aagaard shows that hazard correctness implies the widely accepted Burch-Dill flushing correctness criterion. Windley and Coe [1995] uses HOL [Gordon and Melham, 1993] to define a general theory for the specification and verification of pipelined microprocessors. Ho et al. [1998] use temporal logic to specify properties about structural and control hazards. They call such properties transmission properties. Their method abstracts the pipeline datapath using generalized OR gates that output the collection of all the input values. Transmission properties are then verified using an off-the-shelf model checker.

Mishra et al. [2002] observe that microprocessor verification is made more challenging due to the adhoc creation of the specification model, usually by reverse engineering the RTL design. They propose a top down approach whereby a ‘golden model’ is created in an architecture description language (ADL). The ADL specifies how the implementation should handle hazards: e.g. by stalling the pipeline or by restoring the program counter. The ADL model is then used to generated state machine like specifications against which which they verify the control circuitry. A similar approach is described in Higgins and Aagaard [2005].

Due to the fact that pipeline hazards manifest when multiple instructions interact in the pipeline, coverage of such scenarios using automated test generation is challenging since crafting a suitable sequence of instructions must take into account instruction dependencies, latency through the execution units and the structure of the pipeline. Iwashita et al. [1994] propose a methodology for test case generation using symbolic image computation, similar to reachability analysis in model checking. They first perform a reduction of the pipelined design with respect to the type of properties mentioned in the test cases. The abstract model has fewer instruction types, and only the latency of execution units is preserved. Symbolic reachability on the abstract model is used to derive input sequences for the original design. Gupta et al. [1997] analyze the conditions under which a test generation approach based on an abstract model achieves coverage of the original design. Diep and Shen [1995] use architectural annotations to compute all possible read-after-write (RAW) hazard

in a pipeline. The need for automated test case generation is further advocated with the use of the SPEC benchmarks that is found to achieve poor coverage of the hazards. A similar approach using ADL specifications is taken by Mishra and Dutt [2004], Kohno and Matsumoto [2001].

2.4.3 Datapath Abstraction

There are several approaches to datapath abstraction in formal verification. Exact abstractions [Hojati and Brayton, 1995, Namjoshi and Kurshan, 2000] exploit certain desirable conditions on the transformations and predicates applied to the data values by the control circuitry. The conditions under which the abstraction is exact are identified by syntactic examination of the circuit. Approximate abstractions vary in the degree of the precision they provide. Ho et al. [1998] replace the datapath with union gates that preserve the propagation of inputs into outputs but nothing else. Data predicates become non-deterministic. Datapaths are abstracted with uninterpreted functions and are incrementally improved using counterexamples in Andraus et al. [2006]. Other approaches [Paruthi et al., 1998, Zaraket et al., 2005] aim at preserving the data predicates but restrict the data domain to representative values.

Hojati and Brayton [1995] identify sufficient separation conditions between datapath and control to ensure that the datapath can be abstracted exactly. When the datapath is restricted to perform only data movement operations, the controller is called data independent. Data independent controllers cannot examine any data predicates. With data independent controllers it is sound to reduce datapath variables to a single bit. An extension that generalizes previous work [Norris IP and Dill, 1996] is to allow both data movement and data equality tests. In their terminology, the circuit is said to have a data comparison controller. Hojati and Brayton prove negative results for their model in the case when more than equality tests is allowed.

Bjesse [2008] partitions a word-level netlist into subnets that behave as data comparison controllers. Data-comparison controller nodes (e.g. , equality, multiplexers, etc.) are left as word-level operators. Sub-word operators (e.g. , concatenation, extraction with constant indices, etc.) are decomposed into sub-words. All other operators are exploded into bit-level operations.

A method that generalizes the approach by Hojati and Brayton is presented by Namjoshi and Kurshan [2000]. Their technique calculates the transition relation of each predicate that needs to be preserved using Dijkstra's weakest precondition. This in turn leads to the discovery of new predicates that need to be preserved. Rewrite rules are used to decide if a new predicate is expressed using Boolean connectives in terms of the ones discovered previously.

Paruthi et al. [1998] verify datapath and mixed control/datapath properties. They identify three classes of variables: datapath, control, and mixed. Mixed and control variables are preserved. As with Hojati and Brayton, control circuitry is allowed only to move the datapath variables. However,

in this model, the value of datapath variables is allowed to be computed from the value of other variables using arbitrary operators. They use an interval propagation algorithm to reduce the width of numeric signals to the minimum necessary to preserve exactness (i.e. , no underflow or overflow). Their method does not handle loops with a dynamic count of iterations.

Zaraket et al. [2005] describe an abstraction framework based on bisimulation minimization. Their algorithm uses graph optimization heuristics to identify suitable components of a bit-level netlist for which minimization is desirable. The algorithm computes the classes of an input equivalence that preserves bisimilarity of the subcomponent. Filters of combinational logic are placed to restrict inputs to selected representatives of the equivalence classes. Their method reduces the size of the netlist that propagates datapath values to the bitwidth needed to tranfer and store representative values. The minimized circuit still contains the datapath blocks that transform data values.

Ho et al. [1998] are interested in the verification of parcel-flow properties similar to some aspects of our work: loss, duplication, and ordering. The main idea is to replace the datapath circuitry with wires, replace feedback signals from datapath to control with non-deterministic inputs, and then verify properties about how "tokens" travel through the pipeline. They automatically separate datapath and control based on manually selected seed control signals. They use abstract interpretation to show that parcel-flow properties are preserved on the abstracted circuit.

Andraus et al. [2006] use a language of terms with uninterpreted functions. Heuristics are employed to identify datapath variables based on their width. They use an SMT solver to perform the verification and refine their abstraction in a counterexample-guided refinement loop.

The pipeline model that we propose shares similarities with the token net approach [Ho et al., 1998]. Our pipeline model consists of a network of parcel variables and a controller that steers the parcels through the network. In both models, the toplevel movement of the parcels through the network obeys the rules of data independent controllers identified by Hojati and Brayton [1995], i.e. only copying is allowed. The properties that are likely to be verfied using our abstraction methodology are also similar to the ones in the work by Ho et al.. Both our abstraction and theirs are conservative with respect to parcel flow properties. We differ in how we approach datapath abstraction. They perform a syntactic transformation of the circuit by replacing datapaths with token union gates. Datapath predicates in the abstract circuit take nondeterministic values, which can influence parcel flow. Our approach is aimed at preserving datapath predicates and thus less likely to produce false counterexamples.

Our datapath abstraction technique is based on identifying the equivalences classes of parcel values as they move through the pipeline, under the various combinations of datapath predicates. Because the equivalence holds inductively as the parcels transfer through the pipeline this approach bears similarity to that of Namjoshi and Kurshan that uses bisimulation minimization. While their approach works at the program level, ours exploits the pipeline structure of the circuit. One difficulty

in applying the method in Namjoshi and Kurshan [2000] is convergence of the algorithm. Their algorithm starts with a set of predicates to be preserved and then adds further predicates inductively until the next-state value of each predicate is expressible in terms of the current state value of the set of predicates. In the case of pipelined circuits, their method, which uses Dijkstra's weakest precondition, generates predicates that contain in their definition the datapaths to be abstracted. Identifying whether the set of predicates is stable is essentially what our method does.

Chapter 3

Pipeline Models

This chapter presents our model for pipelined circuits. The definition of the pipeline model is given in Section 3.1. In Section 3.2 we instantiate the concepts of simulation and language containment for the verification of control properties of pipeline models. In Section 3.3 we discuss abstract interpretation of pipeline models.

3.1 Pipeline Models

This section describes our formalization of pipelined circuits as pipeline models. Conceptually, a pipeline model consists of a network of parcel variables and datapath instances. In the network, parcels are either copied between variables or transformed by datapath instances. The parcel flow through the network is coordinated by a state machine that represents the control. The datapath instances are modeled as circuits with annotations describing the parcel and control variables. The network of variables and datapath modules is formalized using if-then-else parcel expressions.

Our presentation of concepts is illustrated using a pipelined circuit called *DiffAddMult*. The structure of the circuit is described in Figure 3.1. *DiffAddMult* has one input v_i and two outputs v_{o1} and v_{o2}. It has four combinational datapath instances: Sub, Neg, Add and $Mult$. There are four registers to hold the intermediate parcel values: r_{Neg}, r_{Add}, r_{Mult1} and r_{Mult2}. The input values are tuples of form $\langle i, j, k, \odot \rangle$ and the output values that are produced correspond to the operation $|i - j| \odot k$, where $\odot \in \{+, *\}$. According to their values, inputs to the pipeline follow one of several paths:

- $Sub \rightarrow Neg \rightarrow Add$
- $Sub \rightarrow Neg \rightarrow Mult$

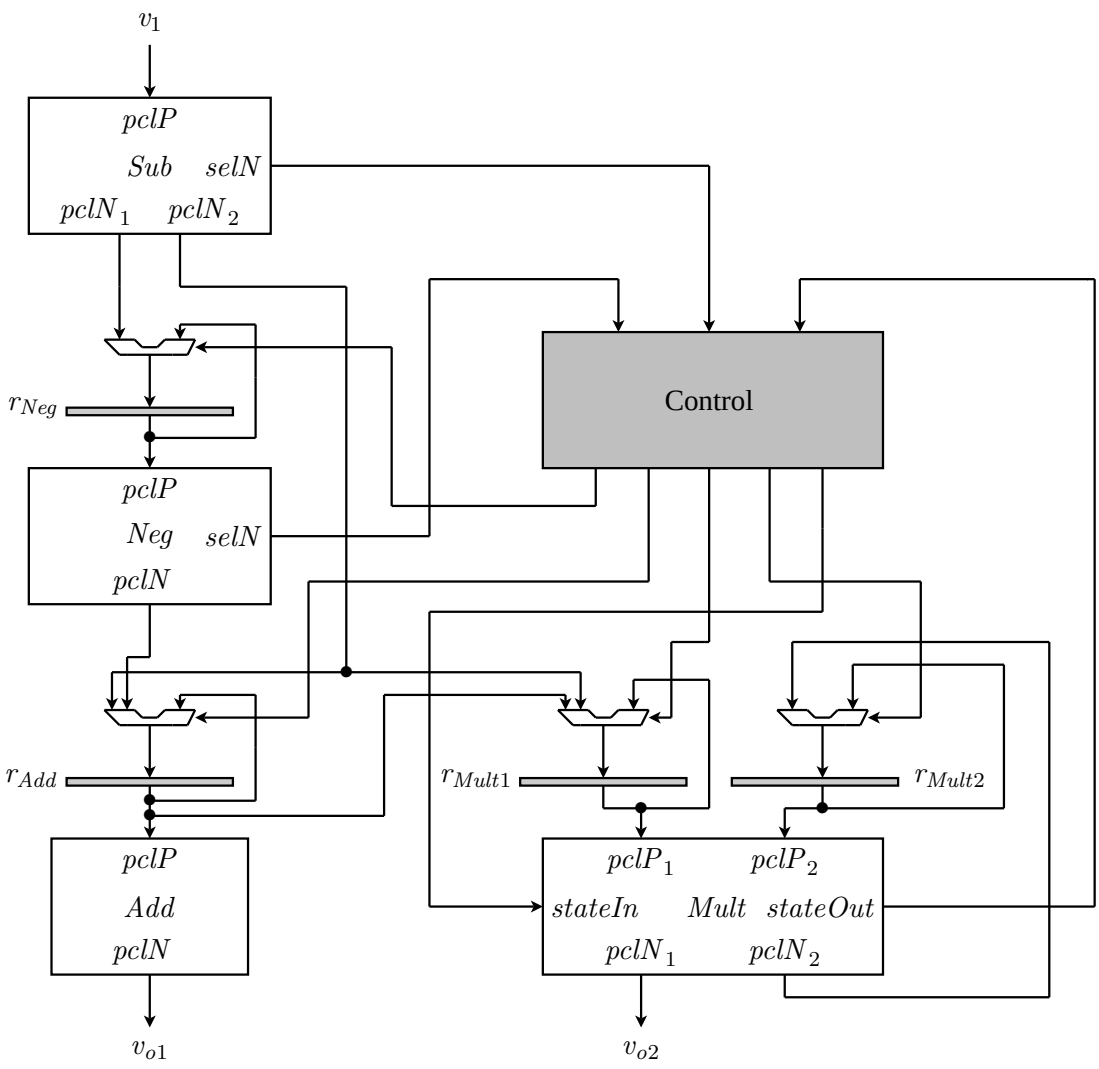


Figure 3.1: Block Diagram of *DiffAddMult*

- $Sub \rightarrow Add$
- $Sub \rightarrow Mult$

The datatypes of an 8-bit version of *DiffAddMult* are described in Figure 3.2.

The Sub instance, described in Figure 3.3 performs the operation $i - j$. The tuple $\langle i, j, k, \odot \rangle$ is encoded in the variable $pclP$. The values i, j are integers represented in 2's complement and k is a natural. The circuit produces two datapath outputs in the variables $pclN_1$ and $pclN_2$. The output $pclN_1$ is meant for the Neg instance and therefore it encodes the operation $\odot$. On the other hand,

```
type op_ty is { add, mult };

type sub_in_ty is
  tuple {
    i : bitvec[8],
    j : bitvec[8],
    k : bitvec[8],
    op : op_ty
  };

type neg_in_ty is
  tuple {
    i : bitvec[8],
    j : bitvec[8],
    op : op_ty
  };

type add_mult_in_ty is
  tuple {
    i : bitvec[8],
    j : bitvec[8]
  };
```

Figure 3.2: *DiffAddMult* data types.

```
1  ckt sub (pclP : sub_in_ty)(pclN_1 : neg_in_ty, pclN_2 : add_mult_in_ty, selN : bitvec[3])
2    var
3      diff : bitvec[8];
4    assign
5      diff := pclP.i − pclP.k;
6      pclN_1 := tuple { i = diff, j = pclP.k, op = pclP.op};
7      pclN_2 := tuple { i = diff, j = pclP.k };
8      selN := if diff < 0 then 001
9                  else if pclP.op = add then 010
10                 else 100;
11 end
```

Figure 3.3: *Sub* Instance

$pclN_2$ is meant to reach directly one of the instances Add or $Mult$ and therefore it does not have to represent $\odot$. The control output $selN$ uses a one-hot representation to encodes the next instance to process the parcel: if $i - j$ is negative the parcel goes to the Neg instance, otherwise to Add or $Mult$ as required by $\odot$. In our example the path of parcels through the pipeline is encoded by the $selN$ outputs of the Sub and Neg instances. The $selN$ signals are used by the control circuitry to drive the muxes of the parcel variables.

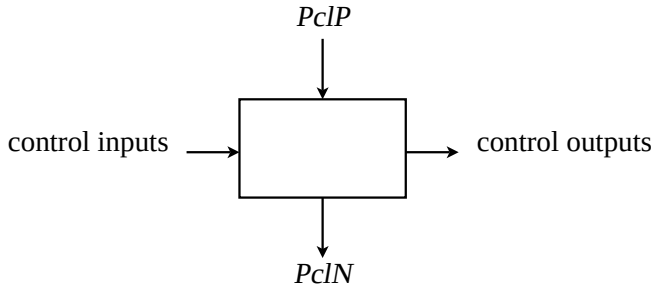


Figure 3.4: Datapath Module

Datapath modules are annotated combinational circuits. The block diagram for a generic datapath module is shown in Figure 3.4. The sets *PclP* and *PclN* denote the input and output parcel variables. The variables in *PclP* and *PclN* are defined by the annotated circuit.

3.1.1 Definition (Datapath Module)**.** A datapath module dp is a tuple $\langle C, \mathit{PclP}, \mathit{PclN}, V_{ctrl} \rangle$ such that:

- C is a combinational circuit
- $\mathit{PclP} \subseteq C.V_i$ is the set of input parcel variables
- $\mathit{PclN} \subseteq C.V_o$ is the set of output parcel variables
- $V_{ctrl} \subseteq C.V_i \cup C.V_o$ is the set of control variables

The datapath module for Sub is defined as follows:

$$\begin{aligned} dp_{Sub} &= \langle C, \mathit{PclP}, \mathit{PclN}, V_{ctrl} \rangle \\ C &= C_{sub} \\ \mathit{PclP} &= \{\, pclP \,\} \\ \mathit{PclN} &= \{\, pclN_1, pclN_2 \,\} \\ V_{ctrl} &= \{\, selN \,\} \end{aligned}$$

```
1  ckt neg (pclP : neg_in_ty)(pclN : add_mult_in_ty, selN : bitvec[2])
2    var
3      abs : bitvec[8];
4    assign
5      abs := 0 − pclP.i;
6      pclN := tuple { i = abs, j = pclP.j };
7      selN := if pclP.op = add then 01 else 10;
8  end
```

$$
\begin{aligned}
dp_{Neg} &= \langle C, PclP, PclN, V_{ctrl} \rangle \\
C &= C_{neg} \\
PclP &= \{\, pclP \,\} \\
PclN &= \{\, pclN \,\} \\
V_{ctrl} &= \{\, selN \,\}
\end{aligned}
$$

Figure 3.5: Neg Datapath Module

The Neg instance is described in Figure 3.5. Its input parcel variable encodes the tuple $\langle (i-j), k, \odot \rangle$, where $i - j < 0$. The resulting output parcel is sent to Add or $Mult$ depending on the operation $\odot$.

```
1  ckt add (pclP : add_mult_in_ty)(pclN : bitvec[8])
2    assign
3      pclN := pclP.i + pclP.j;
4  end
```

$$
\begin{aligned}
dp_{Add} &= \langle C, PclP, PclN, V_{ctrl} \rangle \\
C &= C_{add} \\
PclP &= \{\, pclP \,\} \\
PclN &= \{\, pclN \,\} \\
V_{ctrl} &= \emptyset
\end{aligned}
$$

Figure 3.6: Add Datapath Module

The Add instance is described in Figure 3.6. Its input parcel variable encodes the tuple $\langle |i-j|, k \rangle$. It produces the result of the operation $|i - j| + k$.

The $Mult$ instance is described in Figure 3.7. It works in several stages. The argument $pclP_1$ holds the parcel received from a previous datapath instance and represents the tuple $\langle |i-j|, k \rangle$. The argument $pclP_2$ holds the result of partial products. The operation interprets the operands as

```
1  type mult_pp_ty is tuple { pp1 : bitvec[4], pp2 : bitvec[4] };
2
3  ckt mult (pclP_1 : add_mult_in_ty, pclP_2 : mult_pp_ty, stateIn : bitvec[3])
4                (pclN_1 : bitvec[8], pclN_2 : mult_pp_ty, stateOut : bitvec[3], done : bool)
5    ⋮
6
7  end
```

$$
\begin{aligned}
dp_{Mult} &= \langle C, PclP, PclN, V_{ctrl} \rangle \\
C &= C_{mult} \\
PclP &= \{\, pclP_1, pclP_2 \,\} \\
PclN &= \{\, pclN \,\} \\
V_{ctrl} &= \{\, stateIn, stateOut \,\}
\end{aligned}
$$

Figure 3.7: $Mult$ Datapath Module

2-digit hex numbers and multiplies them according to the standard algorithm, which corresponds to the multiplication of two polynomials: $(b\mathbf{x} + a) \times (d\mathbf{x} + c)$. Each polynomial represents an 8-bit positive number in the form $n = \mathit{high}(n)\mathbf{x} + \mathit{low}(n)$ where $\mathit{high}(n) = n \div 2^4$ and $\mathit{low}(n) = n \mod 2^4$. Multiplying the two polynomials we get $bd\mathbf{x}^2 + (bc + ad)\mathbf{x} + ac$ which represents a 16-bit number. The 8-bit result is thus given by $(\mathit{low}(bc+ad) + \mathit{high}(ac))\mathbf{x} + \mathit{low}(ac)$. Figure 3.8 presents the implementation of the multiplication operation. Figure 3.8a describes the decision tree used to choose the order in which a succession of 4-bit multiplications and additions are performed. The nodes of the decision tree denote the state of the computation. Thus *000* denotes the initial state and *111* denotes the final state. The labels on the edges denote the partial arithmetic operations that are performed and the corresponding condition. The results of the partial operations are returned in $pclN$. The current state and the next state are in the control variables $stateIn$ and $stateOut$. Figure 3.8b represents in more detail the operations performed by each step.

A pipeline model defines two types of variables: parcel and control. The modeling enforces a separation between parcel and control variables according to the view of the pipeline as a network of parcel variables and datapath instances through which parcels flow as steered by the control. Parcel values influence control through the control outputs that datapaths produce.

The set of control variables of a pipeline model is denoted by V_{ctrl}. In the *DiffAddMult* example, the existence of a valid parcel in one of the parcel registers, r_{Neg}, r_{Add}, r_{Mult1}, r_{Mult2}, is represented using correspondingly named control variables: $valid_{Neg}$, $valid_{Add}$, $valid_{Mult1}$, $valid_{Mult2}$. The

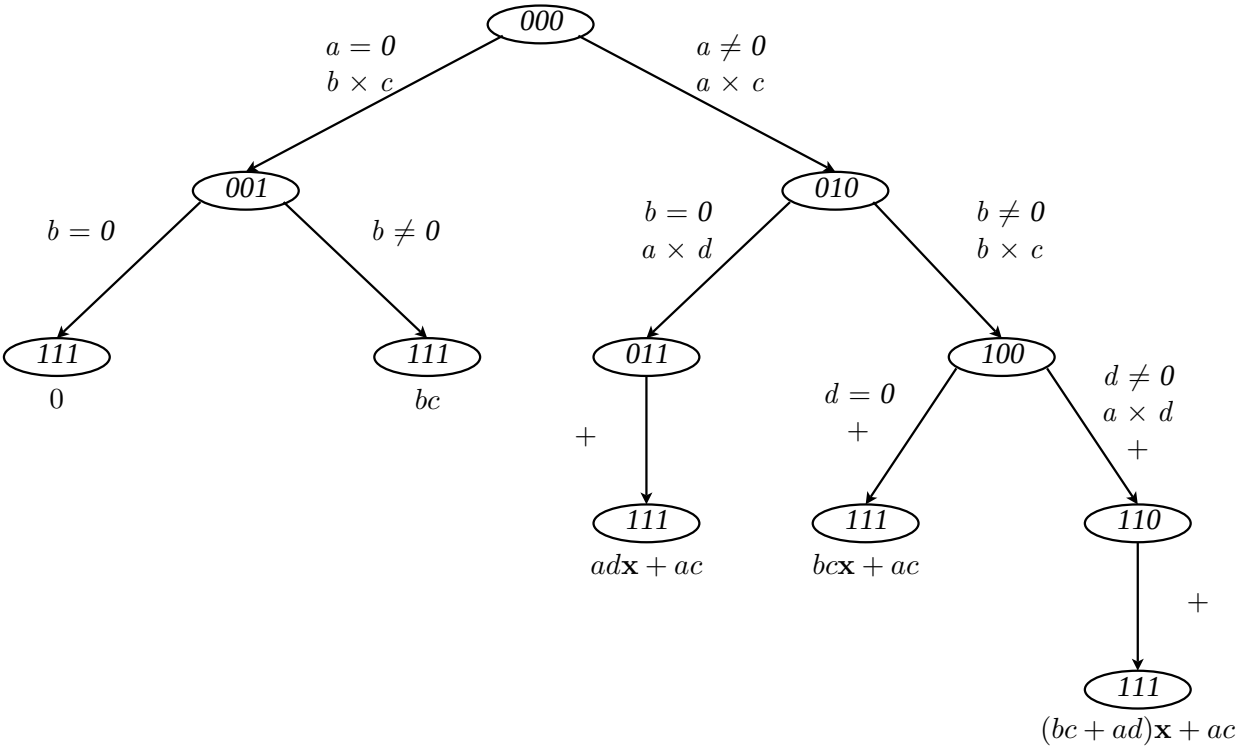


Fig. 3.8a. Decision Tree

State	Condition	Operations	Next State
000	$a = 0$	$b \times c$	*001*
	$a \neq 0$	$a \times c$	*010*
001	$b = 0$		*111*
	$b \neq 0$		*111*
010	$b = 0$	$a \times d$	*011*
	$b \neq 0$	$b \times c$	*100*
011		*low*$(ad) +$ *high*(ac)	*111*
100	$d = 0$	*low*$(bc) +$ *high*(ac)	*111*
	$d \neq 0$	$a \times d \wedge$ *low*$(bc) +$ *high*(ac)	*110*
110		*low*$(ad) +$ $\underbrace{(\textit{low}(bc) + \textit{high}(ad))}_{\text{previous step}}$	*111*

Fig. 3.8b. Transition Table

Figure 3.8: Sequential Multiplication

Variable	Value
$req_{i,Sub}$	input variable
$req_{Sub,Neg}$	$req_{i,Sub} \wedge selN_{Sub} = 00$
$req_{Sub,Add}$	$req_{i,Sub} \wedge selN_{Sub} = 01$
$req_{Sub,Mult}$	$req_{i,Sub} \wedge selN_{Sub} = 10$
$req_{Neg,Add}$	$valid_{Neg} \wedge selN_{Neg} = 0$
$req_{Neg,Mult}$	$valid_{Neg} \wedge selN_{Neg} = 0$
$req_{Add,o1}$	$valid_{Add}$
$req_{Mult,Mult}$	$valid_{Mult1} \wedge stateOut \neq 111$
$req_{Mult,o2}$	$valid_{Mult1} \wedge stateOut = 111$

Table 3.1: Request variables.

Variable	Value
$acc_{Sub,i}$	$req_{i,Sub}$
$acc_{Neg,Sub}$	$req_{Sub,Neg} \wedge \neg stall_{Neg}$
$acc_{Add,Sub}$	$req_{Sub,Add} \wedge \neg(req_{Neg,Add} \vee stall_{Add})$
$acc_{Add,Neg}$	$req_{Neg,Add} \wedge \neg stall_{Add}$
$acc_{Mult,Sub}$	$req_{Sub,Mult} \wedge \neg(req_{Neg,Mult} \vee req_{Mult,Mult} \vee stall_{Mult})$
$acc_{Mult,Neg}$	$req_{Neg,Mult} \wedge \neg(req_{Mult,Mult} \vee stall_{Mult})$
$acc_{Mult,Mult}$	$req_{Mult,Mult}$
$acc_{o1,Add}$	input variable
$acc_{o2,Mult}$	input variable

Table 3.2: Accept variables.

transfer of parcels for *DiffAddMult* is performed through a handshake mechanism. The two parts of the handshake are requests and accepts. Both accepts and requests are modeled using combinational variables. Requests denote where parcels need to transfer in the next cycle and are calculated using control outputs of the datapaths such as the $selN$ output of Sub and Neg or $stateOut$ of $Mult$. Table 3.1 describes the request variables used by *DiffAddMult*. Accepts confirm the request for the transfer of a parcel and the transfer happens in the current cycle. There is an accept-request pair of variables for each edge along which parcels can transfer. Table 3.2 describes the accept variables. To model the stalling of parcels in the parcel registers we use three more variables: $stall_{Neg}$, $stall_{Add}$ and $stall_{Mult}$. Parcels processed by Add or $Mult$ stall if the environment does not accept or in order to satisfy an ordering property by waiting until an older instruction is done. A parcel stalls in Neg if it makes a request that is not granted:

$$stall_{Neg} = (req_{Neg,Add} \wedge \neg acc_{Add,Neg}) \vee (req_{Neg,Mult} \wedge \neg acc_{Mult,Neg})$$

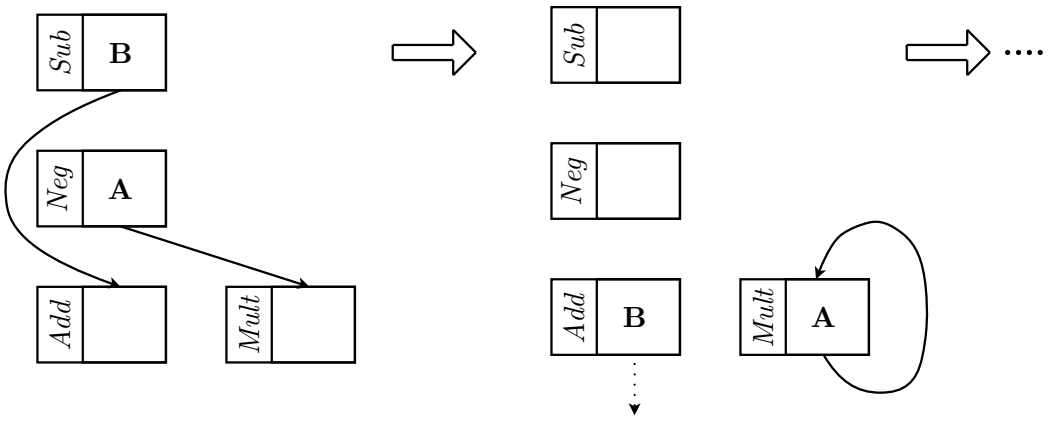


Fig. 3.9a. Both instructions transfer then **B** stalls.

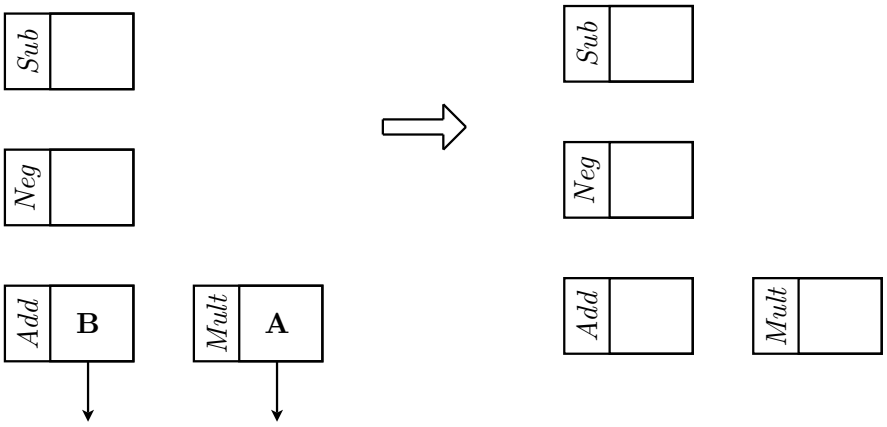


Fig. 3.9b. When **A** is done both instructions exit.

Figure 3.9: **B** waits until **A** finishes processing.

The control circuitry is responsible for ensuring that parcel flow through the pipeline obeys certain desired properties. Figure 3.9 and Figure 3.10 describe pipeline behaviours in which two instructions, **A** and **B**, have a constrained flow. Figure 3.10 describes an ordering property of the *DiffAddMult* pipeline. Instruction **B** enters the pipeline after instruction **A**, in other words, it is younger than **B**, and it must wait for instructions older than it to complete before it can exit the pipeline. Solid arrows denote that requests are made and granted, dotted arrows denote that requests are made but not granted. Figure 3.9a describes a first step in which both instructions transfer which is then followed by a sequence of steps in which instruction **B** is stalled while **A** continues processing. Figure 3.9b describes the step in which both instructions finish processing. Figure 3.10 illustrates

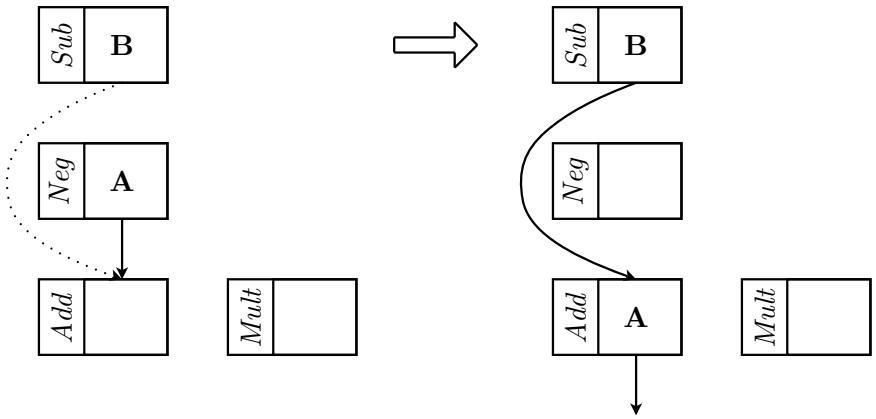


Fig. 3.10a. First cycle: the request of **B** is not granted.

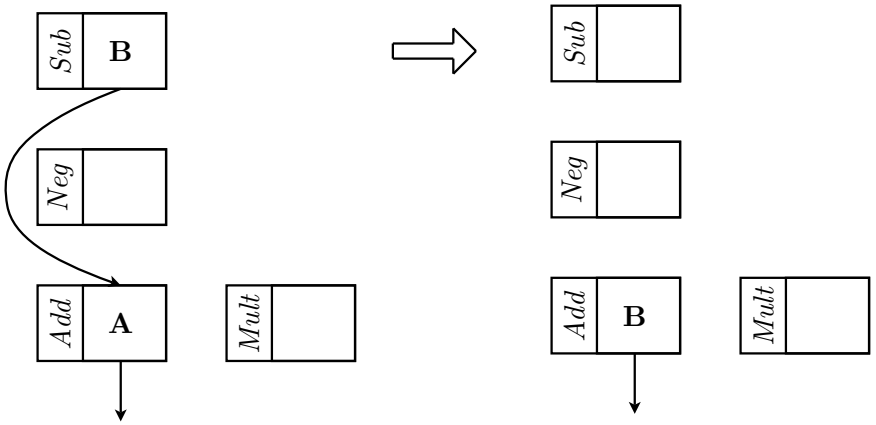


Fig. 3.10b. Second cycle: **B** transfers.

Figure 3.10: **B** stalls because **A** has higher priority.

the prioritization of requests for the Add datapath. Figure 3.10a describes both instructions **A** and **B** requesting to transfer to Add: the request of **B** is not granted and it stalls. Figure 3.10b describes the immediately following cycle when **B** is allowed to tranfer to Add.

We gather the actual parcel arguments to the datapath instances into two sets:

$$\begin{aligned} \mathit{PclP} &\equiv \{\, \mathit{pclP}_{Sub}, \mathit{pclP}_{Neg}, \mathit{pclP}_{Add}, \mathit{pclP}_{Mult1}, \mathit{pclP}_{Mult2} \,\} \\ \mathit{PclN} &\equiv \{\, \mathit{pclN}_{Sub}, \mathit{pclN}_{Neg}, \mathit{pclN}_{Add}, \mathit{pclN}_{Mult1}, \mathit{pclN}_{Mult2} \,\} \end{aligned}$$

For *DiffAddMult* the set of parcel variables is described by

$$V_{pcl} \equiv \{\, v_i, v_{o1}, v_{o2}, r_{Neg}, r_{Add}, r_{Mult1}, r_{Mult2} \,\} \cup \mathit{PclP} \cup \mathit{PclN}$$

We use the following notation for the interesting subsets of parcel variables. The set of constants that appear in if-then-else expressions are denoted by *ConstantPcl.*

Set	Definition	Meaning
CombPcl	$\{\, v_i, v_{o1}, v_{o2} \,\} \cup \mathit{PclP} \cup \mathit{PclN}$	Combinational Variables
RegPcl	$\{\, r_{Neg}, r_{Add}, r_{Mult1}, r_{Mult2} \,\}$	Register Variables
NextRegPcl	$\{\, r_{Neg}{}', r_{Add}{}', r_{Mult1}{}', r_{Mult2}{}' \,\}$	Next State Variables
InputPcl	$\{\, v_i \,\}$	Input Parcel Variables
OutputPcl	$\{\, v_{o1}, v_{o2} \,\}$	Output Parcel Variables
ConstantPcl	$\{\, reset_{Mult} \,\}$	Parcel Constants

Because of the separation between datapath and control, the value of a non-input parcel variable is updated using only the value of another parcel variable. The expressions that can be assigned to parcel variables are called if-then-else (ITE) parcel expressions. An ITE parcel expression is identical to a mux tree, the nodes of which represent parcel variables and with select signals given by expressions over control variables. The simplest type of ITE parcel expressions are parcel variables, constant and $\mathbf{choice}$ expressions. Inductively, if b is a Boolean control expression and t_1 and t_2 are ITE parcel expressions, then $\mathbf{if}\ b\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$ is an ITE parcel expression.

Let *BExpr*(V_{ctrl}) denote the set of Boolean expressions over the set of control variables V_{ctrl}.

3.1.2 Definition (If-then-else Parcel Expressions)**.** The set of if-then-else parcel expressions over V_{pcl} and *BExpr*(V_{ctrl}), denoted by *ITEParcelExpr*(V_{pcl}, V_{ctrl}), is defined inductively:

- If $v \in V_{pcl}$ then v is an ITE parcel expression.
- If $w \in \mathbf{B}^+$ then w is an ITE parcel expression.
- $\mathbf{choice}$ is an ITE parcel expression.
- If t_1 and t_2 are parcel expressions and $b \in$ *BExpr*(V_{ctrl}) then $\mathbf{if}\ b\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$ is a parcel expression.

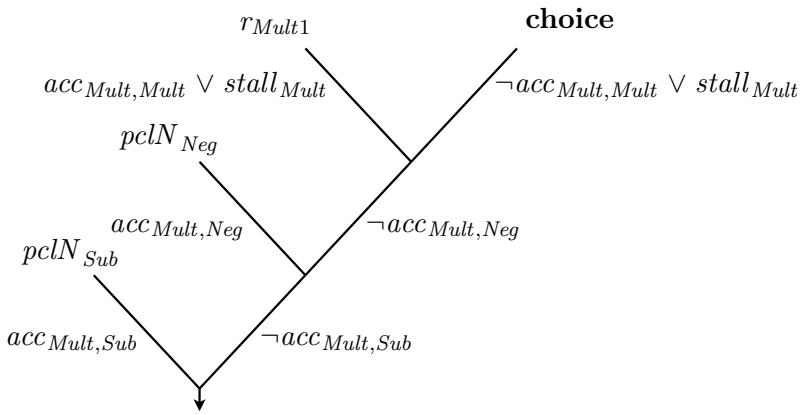


Figure 3.11: Mux tree corresponding to the expression in Equation 3.1.

An example for the expression

$$\begin{aligned}
&\textbf{if } acc_{Mult,Sub} \textbf{ then } pclN_{Sub} \\
&\textbf{else if } acc_{Mult,Neg} \textbf{ then } pclN_{Neg} \\
&\textbf{else if } acc_{Mult,Mult} \vee stall_{Mult} \textbf{ then } r_{Mult1} \\
&\textbf{else choice}
\end{aligned} \tag{3.1}$$

is shown in Figure 3.11. In the figure, each internal node corresponds to the occurrence of an *if-then-else* operator. The leafs of the tree are either parcel variables or the nondeterministic **choice** operator. The edges of the tree are labeled by the conditions under which the corresponding subtree is selected.

The transition relation of *DiffAddMult* contains the following assignments to parcel variables:

$$pclP_{Sub} := v_i \tag{3.2}$$

$$r_{Neg}{}' := \textbf{if } acc_{Neg,Sub} \textbf{ then } pclN_{Sub} \textbf{ else if } stall_{Neg} \textbf{ then } r_{Neg} \textbf{ else choice} \tag{3.3}$$

$$pclP_{Neg} := r_{Neg} \tag{3.4}$$

$$\begin{aligned}
r_{Add}{}' := \;&\textbf{if } acc_{Add}Sub \textbf{ then } pclN_{Sub} \\
&\textbf{else if } acc_{Add,Neg} \textbf{ then } pclN_{Neg} \\
&\textbf{else if } stall_{Add} \textbf{ then } r_{Add} \\
&\textbf{else choice}
\end{aligned} \tag{3.5}$$

$$pclP_{Add} := r_{Add} \tag{3.6}$$

$$v_{o1} := pclN_{Add} \tag{3.7}$$

$$pclP_{Mult1} := r_{Mult1} \tag{3.8}$$

$$
\begin{aligned}
r_{Mult1}{}' &:= \mathbf{if}\ acc_{Mult,Sub}\ \mathbf{then}\ pclN_{Sub} && (3.9)\\
&\quad \mathbf{else\,if}\ acc_{Mult,Neg}\ \mathbf{then}\ pclN_{Neg}\\
&\quad \mathbf{else\,if}\ acc_{Mult,Mult} \vee stall_{Mult}\ \mathbf{then}\ r_{Mult1}\\
&\quad \mathbf{else\,choice}\\
v_{o2} &:= pclN_{Mult} && (3.10)\\
pclP_{Mult2} &:= r_{Mult2} && (3.11)\\
r_{Mult2}{}' &:= \mathbf{if}\ acc_{Mult,Mult}\ \mathbf{then}\ pclN_{Mult2} && (3.12)\\
&\quad \mathbf{else\,if}\ stall_{Mult}\ \mathbf{then}\ r_{Mult2}\\
&\quad \mathbf{else\,if}\ acc_{Mult,Sub} \vee acc_{Mult,Neg}\ \mathbf{then}\ reset_{Mult}\\
&\quad \mathbf{else\,choice} && (3.13)
\end{aligned}
$$

When registers do not hold valid parcel values they are assigned nondeterministically, i.e. **choice**. When a new parcel transfers in the $Mult$ stage the partial product register is reset to a constant value.

A pipeline model is an annotated circuit with syntactic restrictions. The annotations describe the types of variables of the pipeline and its datapath instances.

3.1.3 Definition (Pipeline Model)**.** A pipeline model $Pipe$ is a tuple $\langle C, V_{pcl}, V_{ctrl}, Dps\rangle$ such that:

- C is a circuit.
- V_{pcl} is the set of parcel variables.
- V_{ctrl} is the set of control variables.
- Dps is the set of datapath modules.

We use $C_{DiffAddMult}$ to denote the circuit for the *DiffAddMult* pipeline. The pipeline model for *DiffAddMult* is the tuple

$$
\begin{aligned}
Pipe_{DiffAddMult} &= \langle C, V_{pcl}, V_{ctrl}, Dps\rangle\\
C &= C_{DiffAddMult}
\end{aligned}
$$

The sets of control and parcel variables are disjoint. Parcel variables are assigned ITE parcel terms, control variables are assigned arbitrary expressions over control variables.

$$
\begin{aligned}
\forall\, \text{`}v := e\text{'} \in C.Tr.\ v \in V_{pcl} \cup V_{pcl}{}' &\implies e \in ITEParcelExpr(V_{pcl}, V_{ctrl})\\
\forall\, \text{`}v := e\text{'} \in C.Tr.\ v \in V_{ctrl} \cup V_{ctrl}{}' &\implies vars\, e \subseteq V_{ctrl}
\end{aligned}
$$

Arguments to datapath parcel inputs and outputs are combinational parcel variables of the pipeline model. Arguments to control inputs of datapaths are control variables.

$$\begin{aligned} \forall\ dp \in \mathit{Dps}.\ \mathit{Arg}(dp.V_{ctrl}) &\subseteq V_{ctrl} \\ \forall\ dp \in \mathit{Dps}.\ \mathit{Arg}(dp.V_{pcl}) &\subseteq V_{pcl} \end{aligned}$$

3.2 Abstraction For Control Properties

In this section we specialize the general concepts of simulation and language containment for the verification of control properties of pipeline models. Simulation and language containment are defined with respect to the control variables of the pipeline model. Language containment is weaker than simulation. However, because language containment between parcel automata carries over to pipeline models we prove separate results for both simulation and language containment. Simulation preserves $\mathbf{ACTL}^*$ properties and language containment preserves **LTL** properties.

The definitions of simulation and language containment that preserve control properties are given with respect to a concrete pipeline model $Pipe_c$ and an abstract one $Pipe_a$. In the remainder of the thesis the concrete pipeline model $Pipe_c$ is subject to our datapath abstraction methodology and the abstract model plays the role of a suitable abstraction. The semantics of the two models are defined by their circuits. We denote the corresponding labeled transitions as follows:

$$\begin{aligned} \mathit{LTS}(Pipe_c.C) &= \langle Q_{Pc}, R_{Pc}, T_{Pc}, I_{Pc} \rangle \\ \mathit{LTS}(Pipe_a.C) &= \langle Q_{Pa}, R_{Pa}, T_{Pa}, I_{Pa} \rangle \end{aligned}$$

Since our abstraction is for control properties, the pipeline models $Pipe_c$ and $Pipe_a$ are to be defined over the same set of control variables.

$$Pipe_c.V_{ctrl} = Pipe_a.V_{ctrl}$$

In the remainder of the thesis, $Pipe = \langle C, V_{pcl}, V_{ctrl}, \mathit{Dps} \rangle$ is a pipeline model with its labeled transition system denoted by:

$$\mathit{LTS}(Pipe.C) = \langle Q_P, R_P, T_P, I_P \rangle$$

3.2.1 Simulation

The concept of pipeline model simulation is an instantiation of general simulation (Definition 2.1.2) to pipeline models by requiring that control variables be preserved in the commuting diagram.

3.2.1 Definition (Pipeline Simulation)**.** A pipeline simulation is a simulation relation $\mathcal{S}_P \subseteq Q_{Pc} \times Q_{Pa}$ that preserves control variables.

- Related states agree on register control variables.

$$\forall\, (q_{Pc}, q_{Pa}) \in \mathcal{S}_P.\ q_{Pc} =_{V_{ctrl}} q_{Pa} \tag{3.14}$$

- Commuting diagrams preserve combinational control variables.

$$\begin{array}{l} \forall\, (q_{Pc}, t_{Pc}, q'_{Pc}) \in R_{Pc}.\ \forall\, q_{Pa} \in Q_{Pa}. \\ \quad (q_{Pc}, q_{Pa}) \in \mathcal{S}_P \Longrightarrow \\ \qquad \exists\, t_{Pa} \in T_{Pa}.\ \exists\, {q_{Pa}}' \in Q_{Pa}. \\ \qquad\quad \begin{array}{ccc} q_{Pa} & \overset{t_{Pa}}{\cdots\!\rightarrow} & {q_{Pa}}' \\ \mathcal{S}_P \uparrow & & \mathcal{S}_P \uparrow \\ q_{Pc} & \overset{t_{Pc}}{\longrightarrow} & {q_{Pc}}' \end{array} \quad \text{with } t_{Pc} =_{V_{ctrl}} t_{Pa} \end{array} \tag{3.15}$$

- The condition on initial states remains unchanged.

$$\forall\, q_{Pc} \in I_{Pc}.\ \exists\, q_{Pa} \in I_{Pa}.\ (q_{Pc}, q_{Pa}) \in \mathcal{S}_P \tag{3.16}$$

We use the notation

$$Pipe_c \preceq_P Pipe_a$$

to denote the existence of a relation satisfying Equation 3.14 up to Equation 3.16.

As an example, consider the concrete pipeline model described in Figure 3.12. The domain of the parcel variables consists of tuples of two bit numbers of form $\langle a, b \rangle$. The two datapaths test the two numbers in their operand for equality and, respectively, inequality. The parcel input is passed unchanged to the output. The concrete model is defined by:

$$\begin{array}{rcl} V_{pcl} & = & \{\, i, r, t, o \,\} \\ V_{ctrl} & = & \{\, v_1, v_2 \,\} \\ \mathit{Dps} & = & \{\, Dp_=, Dp_{\neq} \,\} \end{array}$$

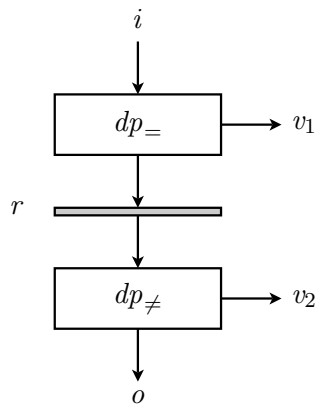


Fig. 3.12a. Block Diagram.

type pcl_ty **is tuple** { **a** : 0 .. 3, **b** : 0 .. 3 };

ckt $Dp_=$($pclP$: pcl_ty)($pclN$: pcl_ty, v : **bool**)
  assign
    $pclN$:= $pclP$;
    v := $pclP$.**a** = $pclP$.**b**;
end

ckt $Dp_\neq$($pclP$: pcl_ty)($pclN$: pcl_ty, v : **bool**)
  assign
    $pclN$:= $pclP$;
    v := **not** ($pclP$.**a** = $pclP$.**b**);
end

ckt $Pipe_c$ (i : pcl_ty)(o : pcl_ty, v_1, v_2 : **bool**)
  var
    r, t : pcl_ty;
  inst
    $dp_=$: $Dp_=$(i)(t,v_1)
    $dp_\neq$: $Dp_\neq$(r)(o,v_2)
  assign
    r' := t;
end

Fig. 3.12b. Implementation

Figure 3.12: Concrete Pipeline Model.

An abstraction for the concrete model is described in Figure 3.13. The abstract model has no input or output parcel variables and the datapath $Dp_{=a}$ produces a nondeterministic parcel output without reading any input. The pipeline model $Pipe_a$ has identically defined control variables, however, the parcel variables differ:

$$\begin{aligned} V_{pcl} &= \{ t, r \} \\ V_{ctrl} &= \{ v_1, v_2 \} \\ Dps &= \{ Dp_{=a}, Dp_{\neq a} \} \end{aligned}$$

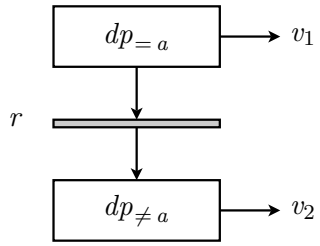


Fig. 3.13a. Block Diagram.

type pcl_ty **is bool**;

ckt $Dp_{=a}$()($pclN$: pcl_ty, v : **bool**)
  assign
    $pclN$:= **choice**;
    v := $pclN$;
end

ckt $Dp_{\neq a}$($pclP$: pcl_ty)(v : **bool**)
  assign
    v := **not** $pclP$;
end

ckt $Pipe_a$ ()(v_1, v_2 : **bool**)
  var
    r, t : pcl_ty;
  inst
    $dp_=$: $Dp_{=a}$()(t,v_1)
    $dp_\neq$: $Dp_{\neq a}$(r)(v_2)
  assign
    r' := t;
end

Fig. 3.13b. Implementation

Figure 3.13: Abstract Pipeline Model.

We define $\mathcal{S}_P$ as follows:

$$\begin{aligned}\mathcal{S}_P \equiv &\{\, (q_{Pc}, q_{Pa}) \mid \exists\, a \in \{\,0, \ldots, 3\,\}.\ q_{Pc}(r) = \langle a, a\rangle \wedge q_{Pa}(r) = \mathbf{true}\,\} \\ &\cup \{\, (q_{Pc}, q_{Pa}) \mid \exists\, a \in \{\,0, \ldots, 3\,\}.\ \exists\, b \in \{\,0, \ldots, 3\,\}.\ q_{Pc}(r) = \langle a, b\rangle \\ &\quad \wedge\, q_{Pa}(r) = \mathbf{false} \wedge a \neq b\,\}\end{aligned} \tag{3.17}$$

The commuting diagram in Equation 3.15 is shown to hold for our example for each of the two cases that define $\mathcal{S}_P$ in Equation 3.17. The case when $q_{Pc}(r) = \langle a, a\rangle$ and $q_{Pa}(r) = \mathbf{true}$ is described in Figure 3.14. There are two types of transitions that the concrete pipeline can make from such a state. In each of the two, there exists a corresponding transition of the abstract model. Figure 3.15 describes how the transitions of a concrete state of form $q_{Pc}(r) = \langle a, b\rangle$ with $a \neq b$ are matched by the simulating abstract state with $q_{Pa}(r) = \mathbf{false}$.

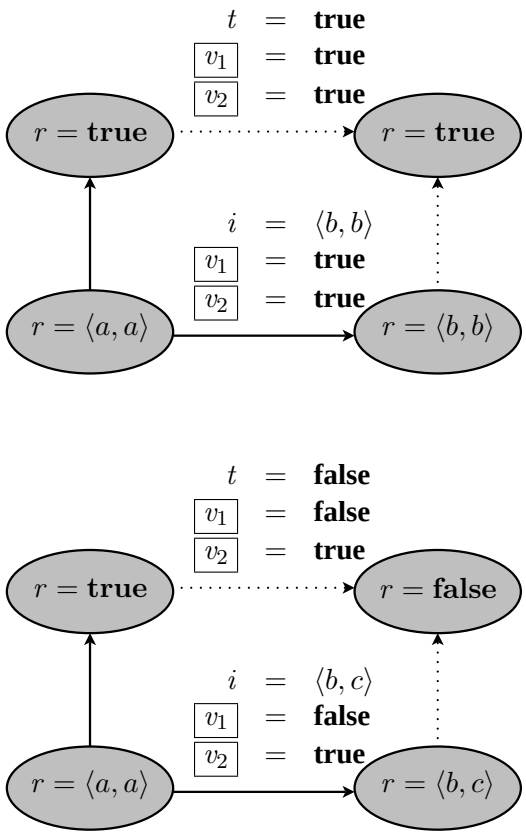


Figure 3.14: Commuting diagrams for the case $q_{Pc}(r) = \langle a, a \rangle$.

3.2.2 Language Containment

A run of the pipeline model $Pipe$ is a run of *LTS*$(Pipe.C)$, as described in Section 2.1. We denote such runs by $\sigma_P : \mathbf{N} \to Q_P \times T_P$ and use the notation $\sigma_P(n) = (q_P^n, t_P^n)$. According to the definition of the run, we have:

$$\forall\, n \in \mathbf{N}.\ (q_P^n, t_P^n, q_P^{n+1}) \in R_P$$

The language of a pipeline model is the set of its runs:

$$\mathcal{L}(Pipe) = \{\, \sigma_P \mid \sigma_P \text{ is a run of } Pipe \,\}$$

Equivalence on runs is defined with respect to a concrete model $Pipe_c$ and an abstract one $Pipe_a$.

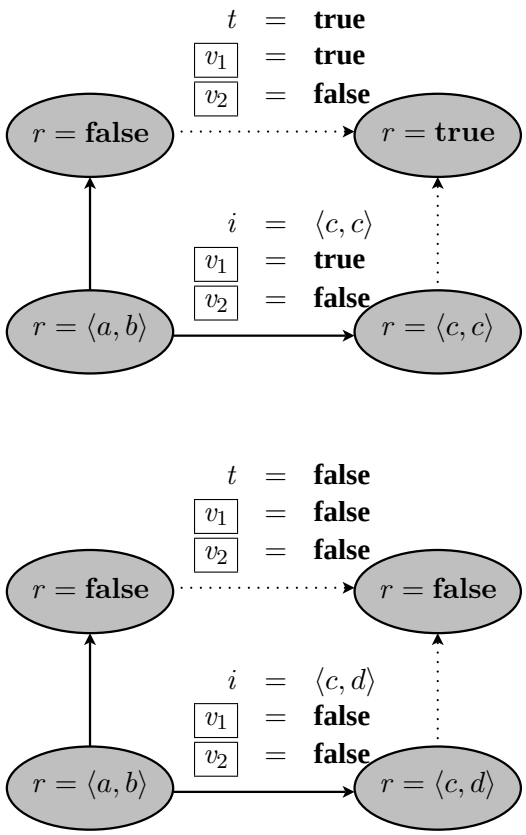


Figure 3.15: Commuting diagrams for the case $q_{Pc}(r) = \langle a, b \rangle$ with $a \neq b$.

Consider $\sigma_{Pc} \in \mathcal{L}(Pipe_c)$ and $\sigma_{Pa} \in \mathcal{L}(Pipe_a)$. Run equality over control variables is denoted by

$$\sigma_{Pc} =_P \sigma_{Pa} \equiv \ \forall\, n \in \mathbf{N}.\ \Big(\ q^n_{Pc} =_{V_{ctrl}} q^n_{Pa} \wedge t^n_{Pc} =_{V_{ctrl}} t^n_{Pa}\ \Big)$$

Language containment of pipeline models is defined by

$$\mathcal{L}(Pipe_c) \subseteq_P \mathcal{L}(Pipe_a) \equiv \forall\, \sigma_{Pc} \in \mathcal{L}(Pipe_c).\ \exists\, \sigma_{Pa} \in \mathcal{L}(Pipe_a).\ \sigma_{Pc} =_P \sigma_{Pa}$$

Figure 3.16 describes an abstraction $Pipe_a$ of the concrete pipeline model $Pipe_c$ from Figure 3.12. The abstract pipeline model has the property that $\mathcal{L}(Pipe_c) \subseteq_P \mathcal{L}(Pipe_a)$. However, $Pipe_c \preceq_P Pipe_a$ does not hold because in each of its states, the model described in Figure 3.16 can perform exactly one transition with label $v_1 = b$, for some $b \in \{\, \mathbf{true}, \mathbf{false} \,\}$, since the parcel input to $dp_{=a}$ is a register. The concrete model can perform transitions with both $v_1 = \mathbf{true}$ and $v_1 = \mathbf{false}$, since

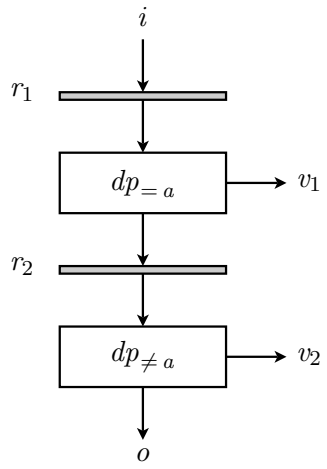


Fig. 3.16a. Block Diagram.

type pcl_ty **is bool**;

ckt $Dp_{=\,a}$($pclP$: pcl_ty)($pclN$: pcl_ty, v : **bool**)
  assign
    $pclN$:= $pclP$;
    v := $pclP$;
end

ckt $Dp_{\neq\,a}$($pclP$: pcl_ty)($pclN$: pcl_ty, v : **bool**)
  assign
    $pclN$:= $pclP$;
    v := **not** $pclP$;
end

ckt $Pipe_{\,a}$ (i : pcl_ty)(o : pcl_ty, v_1, v_2 : **bool**)
  var
    r_1, r_2, t_1, t_2 : pcl_ty;
  inst
    $dp_{=}$: $Dp_{=\,a}$(r_1)(t_1,v_1)
    $dp_{\neq}$: $Dp_{\neq\,a}$(r_2)(o, v_2)
  assign
    ${r_1}'$:= i;
    ${r_2}'$:= t_1;
end

Fig. 3.16b. Implementation

Figure 3.16: Abstract Pipeline Model.

the parcel input to $dp_{=}$ is an input variable of the pipeline model.

In Figure 3.17 we sketch the proof of language containment. The run of the abstract model is chosen on advanced knowledge of the concrete run. The abstract model matches the concrete pipeline step $(q^n_{Pc}, t^n_{Pc}, q^{n+1}_{Pc})$ by choosing state q^n_{Pa} so that $q^n_{Pa}(r_1) = t^n_{Pa}(v_1)$. Because the value of $t^n_{Pc}(v_1)$ is either $\mathbf{true}$ or $\mathbf{false}$ based on whether the input values a^{n+1} and b^{n+1} are equal, while the value of $t^n_{Pa}(v_1)$ is fixed and equal to $q^n_{Pa}(r_1)$, no abstract state can simulate a concrete state.

3.3 Abstract Interpretation Of Pipeline Datapath

Abstract interpretation [Cousot and Cousot, 1977] is an abstraction technique that replaces the concrete data types of a program with abstract data types and the concrete operations with abstract

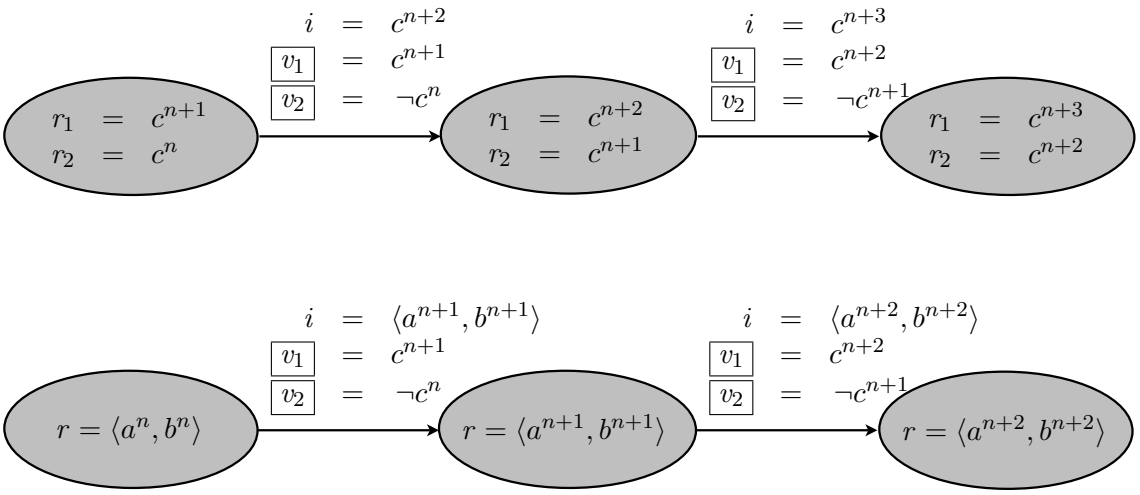


Figure 3.17: Proof Of Language Containment.

ones.

In this section we present the general form of abstract pipeline models created by our abstraction methodology. Abstractions of the concrete pipeline models have the same structure as the concrete ones. By structure we mean variable names and variable use in expressions. Abstract pipelines instantiate abstract datapaths and therefore the domain of the parcel variables is allowed to change. Intuitively, abstract interpretation of a circuit corresponding to a pipeline model allows only for the modification of the declared types of the parcel variables, the name of the instantiated datapath modules and the replacement of a constant's occurrence in an ITE term with the occurrence of another.

We define the equivalence '$\approx_{ai}$' on ITE parcel expressions so that two ITE parcel expressions are equivalent if they differ only by occurrences of constants:

- $e \approx_{ai} e$ for any ITE expression.
- If e_1 and e_2 are constants then $e_1 \approx_{ai} e_2$.
- If $e_1 \approx_{ai} e_3$, $e_2 \approx_{ai} e_4$ and $b \in \mathit{BExpr}(V_{ctrl})$ then

$$(\mathbf{if}\ b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) \approx_{ai} (\mathbf{if}\ b\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4)$$

For the remainder of the thesis we use the following notation for the concrete and abstract datapaths:

$$\begin{aligned} Dps_c &\equiv Pipe_c.\mathit{Dps} \\ Dps_a &\equiv Pipe_a.\mathit{Dps} \end{aligned}$$

3.3.1 Definition (Abstract Interpretation)**.** We say $Pipe_a$ is an abstract interpretation of $Pipe_c$, denoted by

$$Pipe_a = Pipe_c\left[Dps_a / Dps_c\right]$$

if the following conditions hold:

1. The two pipeline models have the same control and parcel variables.

$$\begin{aligned} Pipe_a.V_{ctrl} &= Pipe_c.V_{ctrl} \\ Pipe_a.V_{pcl} &= Pipe_c.V_{pcl} \end{aligned}$$

2. There exists a bijective mapping $\phi : Pipe_a.C.\mathit{Insts} \rightarrow Pipe_c.C.\mathit{Insts}$ such that

$$\begin{aligned} &\forall\, inst \in Pipe_a.C.\mathit{Insts}. \\ &\qquad inst.\mathit{id} = \phi(inst).\mathit{id} \quad \wedge \\ &\qquad inst.\mathit{InputArg} = \phi(inst).\mathit{InputArg} \quad \wedge \\ &\qquad inst.\mathit{OutputArg} = \phi(inst).\mathit{OutputArg} \end{aligned}$$

3. There exists a bijective mapping $\psi : Pipe_a.C.\mathit{Tr} \rightarrow Pipe_c.C.\mathit{Tr}$ such that

$$\begin{aligned} &\forall\, \text{`}v := e\text{'} \in Pipe_a.C.\mathit{Tr}. \\ &\quad v \in V_{ctrl} \cup {V_{ctrl}}' \Longrightarrow \psi(\text{`}v := e\text{'}) = \text{`}v := e\text{'} \\ &\qquad \wedge \\ &\quad v \in V_{pcl} \cup {V_{pcl}}' \Longrightarrow \exists\, e'.\ e \approx_{ai} e' \wedge \psi(\text{`}v := e\text{'}) = \text{`}v := e'\text{'} \end{aligned}$$

4. Initial conditions on control variables are the same.

$$\forall\, v \in V_{ctrl}.\ \text{`}v := e\text{'} \in Pipe_c.C.\mathit{Init} \Longleftrightarrow \text{`}v := e\text{'} \in Pipe_a.C.\mathit{Init}$$

If $Pipe_a = Pipe_c\left[Dps_a / Dps_c\right]$ then the circuits of the two pipelines assign the same variables. Therefore, any $v \in V_{pcl} \cup V_{ctrl}$ is either combinational in both circuits or register in both.

Consider the concrete pipeline model described in Figure 3.12. The abstract pipeline model in Figure 3.13 is not an abstract interpretation because it does not have the parcel variables i and o.

On the other hand, the abstract model in Figure 3.16 is a proper abstract interpretation.

3.4 Summary

We describe the model of pipelined circuits as a network of parcel variables and datapath instances through which parcels flow as coordinated by the control circuitry. The variables of the circuit are divided into datapath and control. The separation is enforced by syntactic restrictions on the type of expressions that can be assigned to each of the two kinds of variables. Abstract interpretation of the datapath is performed by replacing the concrete datapaths by abstract ones. The type of the pipeline parcel variables is adjusted accordingly. The control is left unchanged.

Chapter 4

Parcel Automata

In this chapter we present a computation model of the pipeline datapath. The model for the pipeline datapath is a labeled transition system, called a parcel automaton, that describes the behaviour of the pipeline datapath with respect to the control inputs and outputs of the datapath instances as parcels move through the pipeline. Abstractions of parcel automata are defined using simulation or language containment and are shown to preserve the control visible behavior of the datapath. Abstract parcel automata are used to define abstract datapaths.

A parcel represents a group of related values which are held in parcel variables during a pipeline computation. Both the values of the parcel and the corresponding variables change during the computation of the pipeline model. In a particular pipeline step, the parcel is identified by its variables, which can be both register and combinational. We define parcels as non-empty subsets of $V_{pcl} \cup$ *NextRegPcl.*

During a pipeline computation, the parcel propagates through parcel variables and datapaths. This execution trace is called a parcel computation. The parcel computation records the transformation of the parcel's variables and its interaction with the control circuitry.

In a pipeline computation, multiple parcel computations take place simultaneously. A very important characteristic of the parcel computations that coexist during a computation of the pipeline model is that within a pipeline step, they do not share parcel variables or datapaths. This property of pipeline computations is called parcel independence and is formalized in the next chapter.

Section 4.1 describes a complete example for using parcel automata for datapath abstraction. In Section 4.2 we describe fan-out graphs which model the propagation of a parcel's variables through combinational variables and datapaths into next state registers. Section 4.3 illustrates parcel computations with the *DiffAddMult* example. The definition of parcel automata is given in Section 4.5.

Section 4.6 adapts the concepts of simulation and language containment to parcel automata. Section 4.7 shows how abstract interpretation is performed using abstract parcel automata.

4.1 Overview Of Abstraction Using Parcel Automata

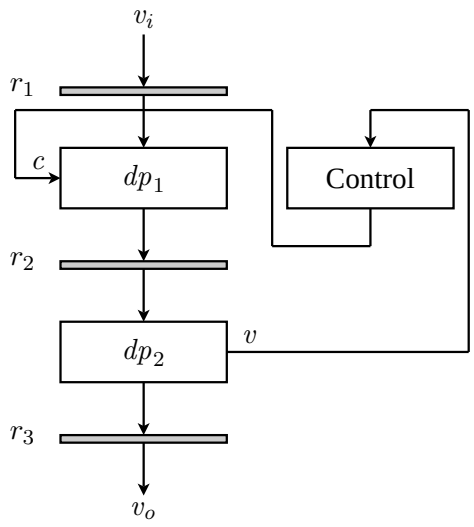


Figure 4.1: $AndOr$ Block Diagram.

We introduce a simple example called $AndOr$ and use it to provide a high-level description of the methodology of using parcel automata for datapath abstraction. Our example describes a pipeline computation and the contained parcel computations, a parcel automaton that models all the possible parcel computations, and an abstract parcel automaton and an abstract pipeline model obtained by abstract interpretation using the abstract parcel automaton.

The $AndOr$ pipeline consists of three parcel registers and two datapaths as shown in Figure 4.1. There are two control variables, a two-bit register c and a one-bit combinational variable v. The implementation of the circuit is given in Figure 4.2. The first datapath, Dp_1, is described on lines 1–7. Its input arguments are a two-bit input parcel and a two-bit control variable. It produces an output parcel that consists of four bits obtained by the concatenation of four bit-and operations. The second datapath, Dp_2, is described on lines 9–13. The resulting two-bit parcel value is the bit-or of the two-bit halves of its input parcel. Dp_2 produces a one-bit control output that consists of the bit-or of the two bits of the output parcel. The pipeline circuit instantiates the two datapaths. The state of the control circuitry consists of the two-bit register c. Its value is updated using the control output of Dp_2. The current value of c is the control argument provided to Dp_1.

```
1  ckt Dp1(pclP : bitvec[2], v : bitvec[2])(pclN : bitvec[4])
2    assign
3      pclN[0] := pclP[0] and v[0];
4      pclN[1] := pclP[1] and v[0];
5      pclN[3] := pclP[0] and v[1];
6      pclN[4] := pclP[1] and v[1];
7  end
8
9  ckt Dp2(pclP : bitvec[4])(pclN : bitvec[2], v : bitvec[1])
10   assign
11     pclN := pclP[0:1] or pclP[2:3];
12     v := pclN[0] or pclN[1];
13 end
14
15 ckt Pipe_c (v_i : bitvec[2])(v_o : bitvec[2])
16   var
17     v : bitvec[1];
18     r1, r3, c, pclP1, pclN2 : bitvec[2];
19     r2, pclN1, pclP2 : bitvec[4];
20   inst
21     dp1 : Dp1(pclP1,c)(pclN1);
22     dp2 : Dp2(pclP2)(pclN2,v);
23   assign
24     pclP1 := r1;
25     pclP2 := r2;
26     c' := v :: (not v);
27     r1' := v_i;
28     r2' := pclN1;
29     r3' := pclN2;
30     v_o := r3;
31   init
32     c := 01;
33     r1 := 00;
34     r2 := 0000;
35     r3 := 00;
36 end
```

Figure 4.2: $AndOr$ Implementation.

An example of a pipeline computation of $AndOr$ is provided in Figure 4.3. The top half of the figure displays a sequence of pipeline model states. Each state is represented by the values of the three parcel registers, highlighed in distinct shades of grey. In the first state, for instance, the values

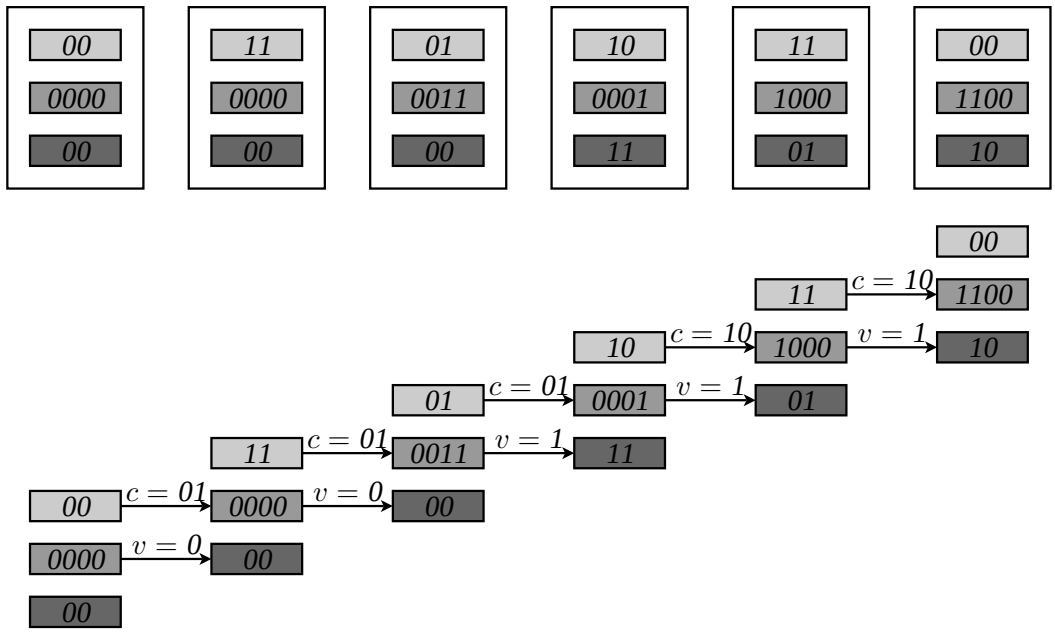


Figure 4.3: $AndOr$ Computation.

of the three registers are:

$$\begin{aligned} r_1 &= 00 \\ r_2 &= 0000 \\ r_3 &= 00 \end{aligned}$$

The bottom half of the figure lays out the states of the pipeline computation on a diagonal, with the effect that from left to right one can trace each parcel through the pipeline. For instance, the parcels in the first state of the computation have the following traces:

$$\begin{array}{lllll} r_1 = 00 & \longrightarrow & r_2 = 0000 & \longrightarrow & r_3 = 00 \\ & & r_2 = 0000 & \longrightarrow & r_3 = 00 \\ & & & & r_3 = 00 \end{array}$$

A more accurate description of the trace of a parcel is to include the control values that influence its computation. For each parcel, there are two such values, c when the parcel passes through Dp_1 and

v when the parcel goes through Dp_2. Two enhanced traces from Figure 4.3 are as follows:

$$r_1 = 00 \xrightarrow{c=01} r_2 = 0000 \xrightarrow{v=0} r_3 = 00$$
$$r_1 = 10 \xrightarrow{c=10} r_2 = 1000 \xrightarrow{v=1} r_3 = 10$$

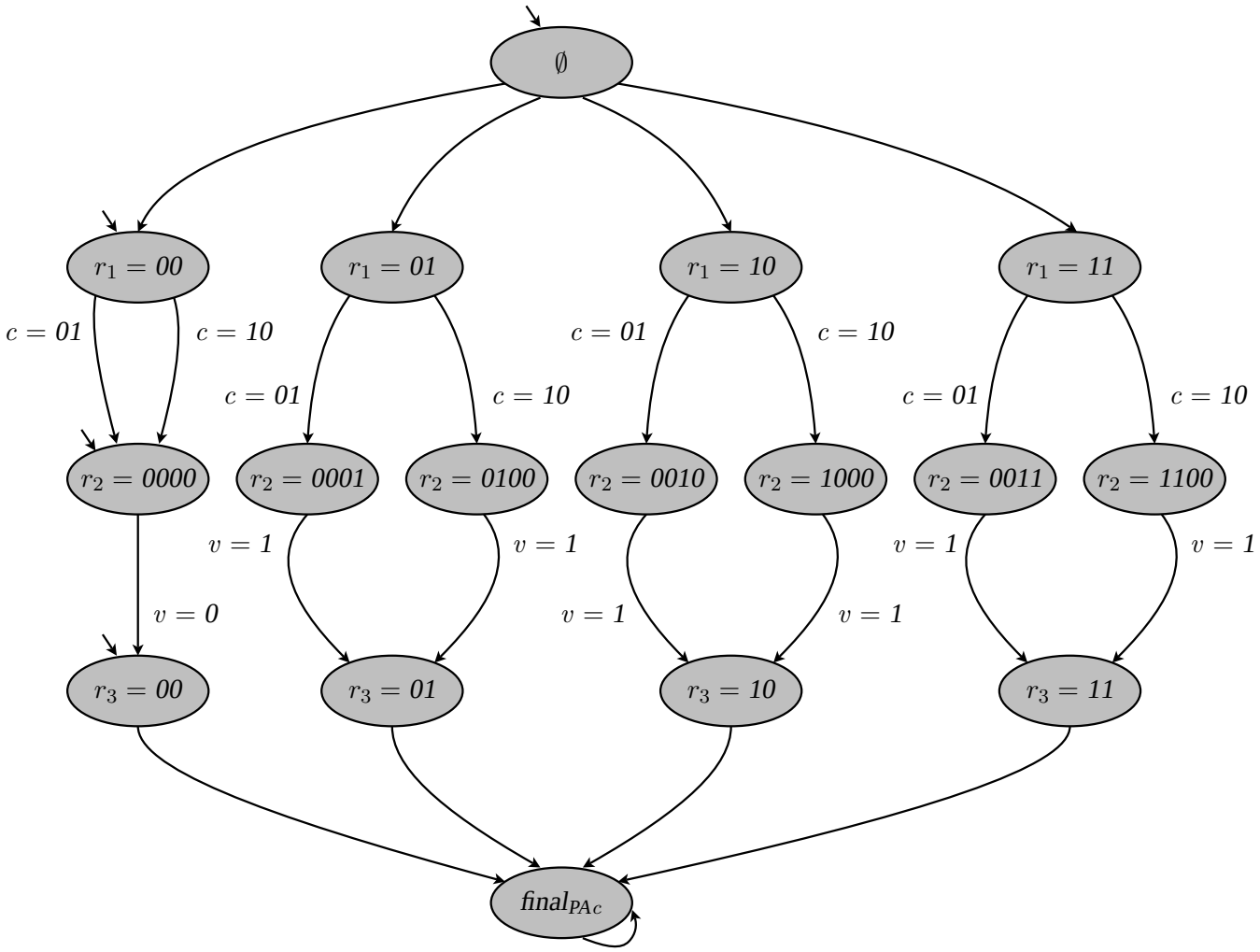


Figure 4.4: $AndOr$ Parcel Automaton.

A parcel automaton models the traces of parcels through the pipeline as a labeled transition system. The states of the automaton correspond to parcel values in given parcel registers and the transitions correspond to the transfer of parcels from one register to the next. The transition labels give the

control values that influence the parcel during a transition. The parcel automaton for $AndOr$ is shown in Figure 4.4. It describes the traces through the pipeline of all possible input parcel values. The empty state $\emptyset$ denotes the state of the parcel before entering the pipeline. The final state final_{PAc} represents the parcel after it has exited the pipeline. After a transition from the empty state, the state of the parcel may be any of the four two-bit values in register r_1. For each such value, and for each value of the control variable c, the automaton makes a transition to a state that represents the parcel in register r_2. From such a state the automaton makes a transition to a state representing the parcel in register r_3. The label of such a transition shows the value of the control variable v because it is produced by the datapath Dp_2 based on the value of parcel. A transition to the final state completes the computation of the parcel automaton.

Datapath abstraction is based on identifying parcel values that have the same control visible behaviour through the pipeline. The parcel automaton representation of the datapath allows us to formulate this problem as a relationship between parcel automaton states. The reduction of the datapath is performed by collapsing together equivalent states of the parcel automaton. In our example, the parcel automaton shows that the states corresponding to three of the input values, *01*, *10* and *11* are equivalent. After the reduction step that preserves only the state $r_1 = \mathit{01}$ and discards the other two equivalent ones, the automaton is shown in Figure 4.5. The reduced automaton has two more equivalent states $r_2 = \mathit{0001}$ and $r_2 = \mathit{0100}$. After performing a second reduction and representing the reduced data domain using the abstract values $\{ \alpha_1, \alpha_2, \beta_1, \beta_2 \}$ we obtain the abstract parcel automaton shown in Figure 4.6.

Our example showed how a parcel automaton is created to represent the datapath computations. Conversely, given an abstract parcel automaton we can derive the pipeline datapath that it represents. Corresponding to the abstract $AndOr$ parcel automaton, the implementations of the abstract datapaths are shown in Figure 4.7. The abstract datapath $Dp_{1\,a}$, shown on lines 4–9, corresponds to the transitions of the parcel automaton from a state with register r_1 to a state with register r_2. $Dp_{1\,a}$ transforms α_1 into α_2 and β_1 into β_2. The value of the control input does not influence its computation. The second abstract datapath is shown on lines 11–19. It, too, encodes two separate transitions of the parcel automaton, corresponding to parcels moving from register r_2 to r_3. Both the value of the parcel output $pclN$ and the control output v are sensitive to the value of the input parcel. The two datapaths are used to give an abstract interpretation of the original pipeline as shown on lines 21–42 of the listing in Figure 4.7. The abstract pipeline circuit differs from the concrete one in the types of parcel variables, the two datapaths that it instantiates and the initial condition. Figure 4.8 shows the abstract computation that is equivalent to the concrete one in Figure 4.3.

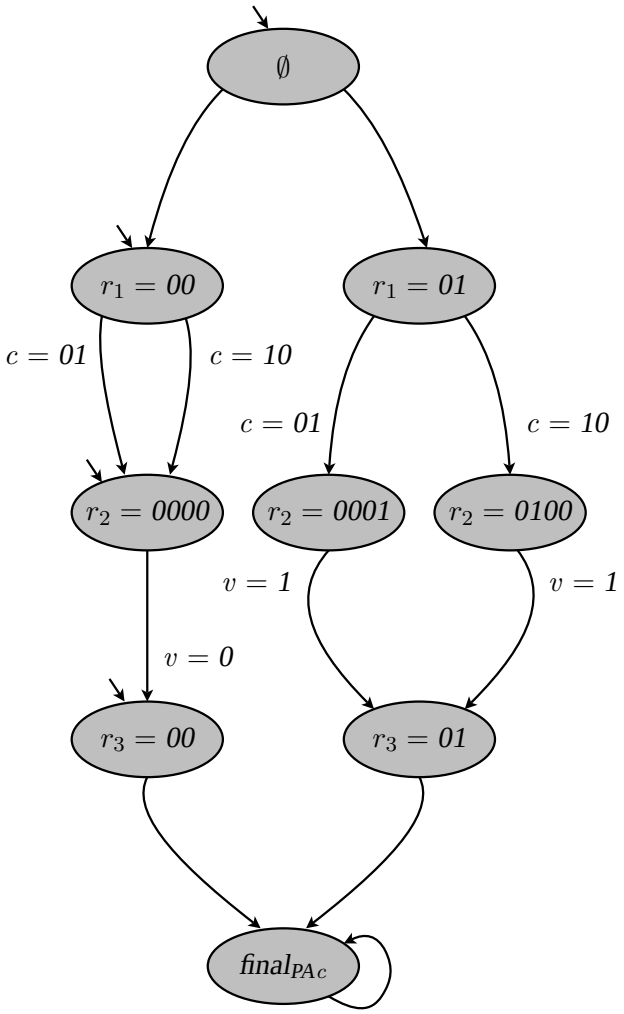


Figure 4.5: $AndOr$ parcel automaton after one reduction step.

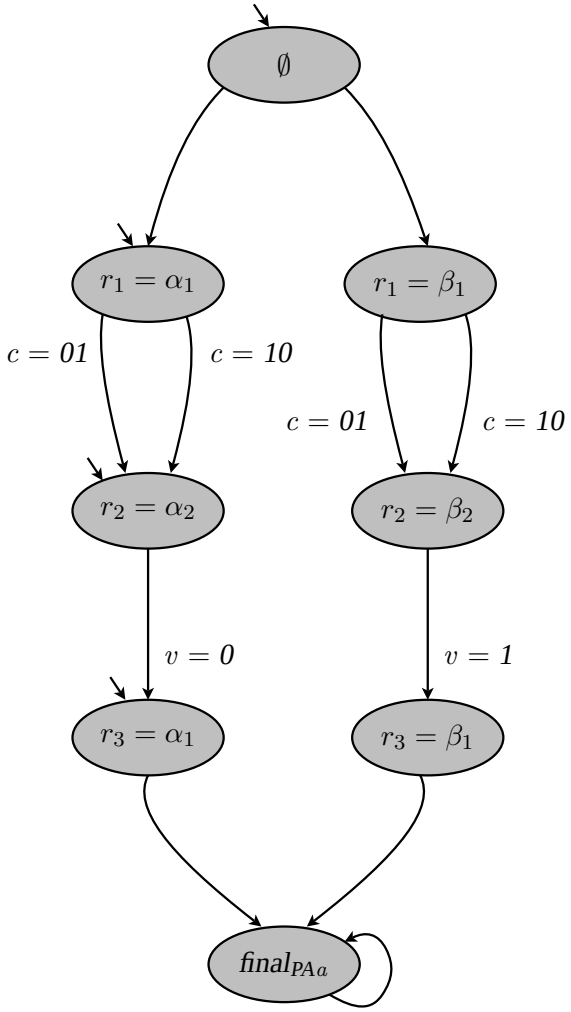


Figure 4.6: $AndOr$ Abstract Parcel Automaton.

```
1  type pcl_ty is { α1, α2, β1, β2 };
2  type pcl_in_ty is { α1, β1 };
3
4  ckt Dp1 a(pclP :pcl_in_ty, v : bitvec[2])(pclN : pcl_ty)
5    assign
6      pclN := if pclP == α1 then α2
7                  else if pclP == β1 then β2
8                  else choice;
9  end
10
11 ckt Dp2 a(pclP : pcl_ty)(pclN : pcl_ty, v : bitvec[1])
12   assign
13     pclN := if pclP == α2 then α1
14                 else if pclP == β2 then β1
15                 else pclP;
16     v := if pclP == α2 then 0
17                 else if pclP == β2 then 1
18                 else choice;
19 end
20
21 ckt Pipe a (vi : pcl_in_ty)(vo : pcl_ty)
22   var
23     v : bitvec[1];
24     r1, r3, c, pclP1, pclN2 : pcl_ty;
25     r2, pclN1, pclP2 : pcl_ty;
26   inst
27     dp1 : Dp1(pclP1,c)(pclN1);
28     dp2 : Dp2(pclP2)(pclN2,v);
29   assign
30     pclP1 := r1;
31     pclP2 := r2;
32     c' := v :: (not v);
33     r1' := vi;
34     r2' := pclN1;
35     r3' := pclN2;
36     vo := r3;
37   init
38     c := 01;
39     r1 := α1;
40     r2 := α2;
41     r3 := α2;
42 end
```

Figure 4.7: $AndOr$ Abstract Implementation.

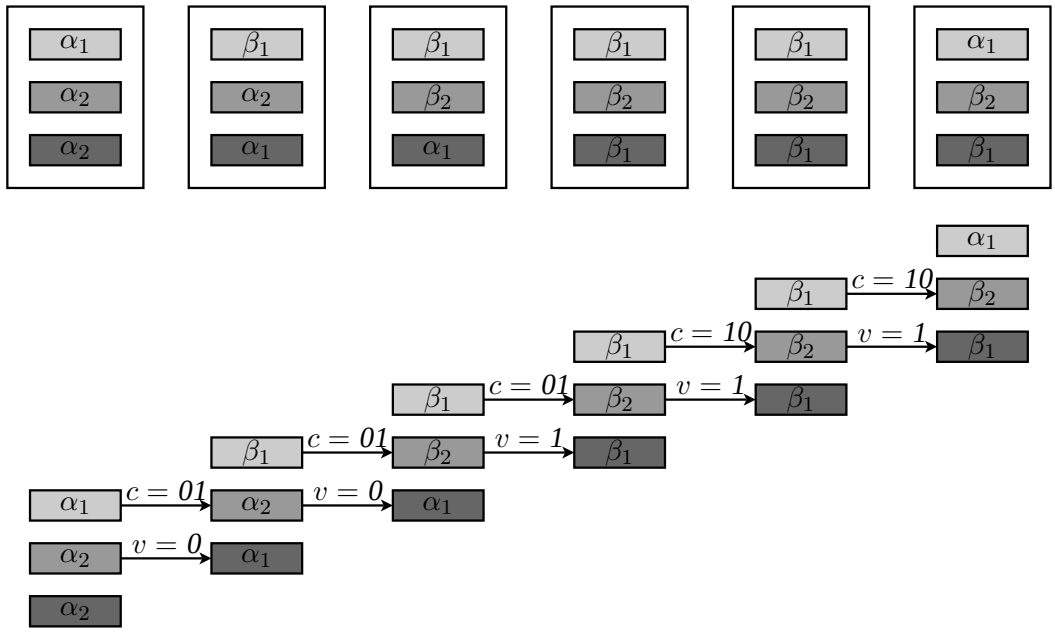


Figure 4.8: Abstract $AndOr$ Computation.

4.2 Fan-Out Graphs

We use fan-out graphs to give a precise definition of the propagation of a parcel's values through parcel variables and datapaths. A fan-out graph is a directed acyclic graph. The nodes of the graph are parcel variables and the edges denote the transfer of a value from the source of the edge to its destination. The labels on the edges of the graph stand for the condition under which the transfer takes place.

4.2.1 Definition (Fan-out Graph)**.** A fan-out graph is a tuple $\langle \mathit{Nodes}, \mathit{Succ} \rangle$ where:

- *Nodes* is the set of nodes.
- *Succ* is the successor relation.
- $\mathit{Nodes} \subseteq \mathit{V}_{pcl} \cup \mathit{NextRegPcl} \cup \mathit{ConstantPcl}$
- $\mathit{Succ} \subseteq \mathit{Nodes} \times \mathit{BExpr}(\mathit{V}_{ctrl}) \times \mathit{Nodes}$ is a set of fan-out edges.

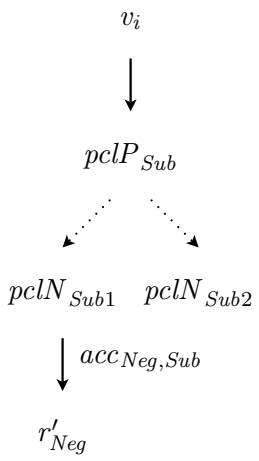


Figure 4.9: Fan-out Graph

Figure 4.9 represents a fan-out graph for the transfer of an input parcel into the register r_{Neg}. Solid edges in the figure represent parcel copying from one variable to another. Dotted lines denote parcel transformations by datapaths. The label on an edge denotes the condition under which the transfer occurs. The omission of a label implies its condition is always true.

Given a fan-out graph *fg* we refer to the datapaths it references by

$$\mathit{datapaths}\ \mathit{fg} \equiv \{\ \mathit{dp} \in \mathit{Dps} \mid \mathit{Arg}(\mathit{dp}.\mathit{V}_{pcl}) \cap \mathit{fg}.\mathit{Nodes} \neq \emptyset\ \}$$

For the example in Figure 4.9 we have *datapaths* $fg = \{ Sub \}$.

The fan-out of a parcel is represented by a fan-out graph *fanOut* $(q_P, t_P, q'_P)\ p$ that has as roots the variables in p and as nodes all variables that derive their value transitively from the variables in the parcel in the pipeline step (q_P, t_P, q'_P). When the pipeline step is known from context, we omit it, and write *fanOut* p.

4.2.2 Definition (Parcel Fan-out)**.** The fan-out graph of a parcel p in (q_P, t_P, q'_P) is the fan-out graph $fg = \langle Nodes, Succ \rangle$ defined inductively:

Base Case

- $p \subseteq Nodes$
- If w is a constant, $v \in p$, and $(w, b, v) \in FanOutEdges$ such that $(q_P \cup t_P) \models b$ then $w \in$ *Nodes*.

Inductive Case If $v_l \in Nodes$ and there exists a fan-out edge (v_l, b, v_k) such that $(q_P \cup t_P) \models b$ then $v_k \in Nodes$ and $(v_l, b, v_k) \in Succ$.

Formally, the roots of the fan-out graph *fg* is the set of variables defined as follows:

$$\mathit{roots}\ fg = \{\ v \in fg.\mathit{Nodes} \mid \nexists\ (v_1, b, v) \in FanOutEdges.\ (v_1, b, v) \in fg.\mathit{Succ}\ \}$$

They are the roots of the fan-out digraph with the addition of the variables that have an incoming edge from a constant.

The variables in the fan-out of a parcel p are denoted by p^*:

$$p^* \equiv (\mathit{fanOut}\ p).\mathit{Nodes} \cap (V_{pcl} \cup \mathit{NextRegPcl})$$

Parcel variables receive their value either by assignment of an ITE parcel expression or by being an actual parameter for an output of a datapath instance. The meaning of an ITE parcel expression is a set of pairs of form $(expr, cond)$, where $expr$ is a parcel variable, a constant or **choice** and $cond$ is a Boolean control expression, with mutually exclusive conditions, such that $expr$ is the value of the expression when $cond$ is true. The support function returns the set of expression-condition pairs for a given expression. Because edges of a fan-out graph are derived using the support of an expression t, the definition of the support function ensures that every variable occurs at most once in the support of an expression.

4.2.3 Definition (Support function)**.** The support of an ITE parcel expression t is defined using the auxiliary function $\mathit{support}_1\ t$. The function $\mathit{support}_1$ is defined inductively:

- $t = w$ with w constant then $\mathit{support}_1\ t = \{\ (w, \mathbf{true})\ \}$.
- $t = \mathbf{choice}$ then $\mathit{support}_1\ t = \{\ (\mathbf{choice}, \mathbf{true}\ \}$.
- If $t = v$ with $v \in V_{pcl}$ then $\mathit{support}_1\ t = \{\ (v, \mathbf{true})\ \}$.
- If $t = \mathbf{if}\ b\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$ then

  $$\mathit{support}_1\ t = \{\ (v, b \wedge b_1) \mid (v, b_1) \in \mathit{support}_1\ t_1\ \} \cup \{\ (v, (\neg b) \wedge b_2) \mid (v, b_2) \in \mathit{support}_1\ t_2\ \}$$

- *support* joins the multiple occurrences of the same expression e in $\mathit{support}_1\ t$:

$$\mathit{support}\ t \equiv \{\ (e, \bigvee_{(e,b) \in \mathit{support}_1\ t} b) \mid \exists\, b_1.\ (e, b_1) \in \mathit{support}_1\ t\ \}$$

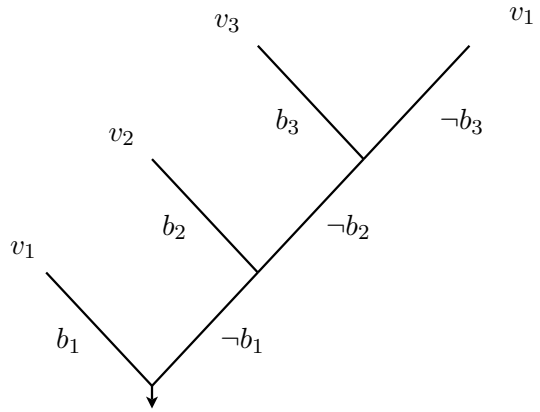


Figure 4.10: Mux tree for the expression $\mathbf{if}\ b_1\ \mathbf{then}\ v_1\ \mathbf{else\ if}\ b_2\ \mathbf{then}\ v_2\ \mathbf{else\ if}\ b_3\ \mathbf{then}\ v_3\ \mathbf{else}\ v_1$.

For the expression

$$t = \mathbf{if}\ b_1\ \mathbf{then}\ v_1\ \mathbf{else\ if}\ b_2\ \mathbf{then}\ v_2\ \mathbf{else\ if}\ b_3\ \mathbf{then}\ v_3\ \mathbf{else}\ v_1$$

described in Figure 4.10, *support* t contains only one occurrence of the variable v_1:

$$(v_1, b_1 \vee \neg b_1 \wedge \neg b_2 \wedge \neg b_3)$$

For the expression in Equation 3.1 on page 44 we have:

$$\begin{aligned} \mathit{support}\ t \quad = \quad & \{\ (pclN_{Sub}, acc_{Mult,Sub}), \\ & (pclN_{Neg}, \neg acc_{Mult,Sub} \wedge acc_{Mult,Neg}), \end{aligned}$$

$$(r_{Mult1}, \neg acc_{Mult,Sub} \wedge \neg acc_{Mult,Neg} \wedge (acc_{Mult,Mult} \vee stall_{Mult})) \}$$

Since accepts or stalls are mutually exclusive the example above simplifies to:

$$\mathit{support}\, t = \{ (pclN_{Sub}, acc_{Mult,Sub}), (pclN_{Neg}, acc_{Mult,Neg}), (r_{Mult1}, acc_{Mult,Mult} \vee stall_{Mult}) \}$$

Fan-out graphs are inferred from assignments to parcel variables in the transition relation and from the arguments of parcel inputs and outputs of the datapath instances. Corresponding to each such case we have a type of fan-out edge. The set of fan-out edges is denoted by $FanOutEdges$.

Assignments

$$(e, b, v_k) \in FanOutEdges \Longrightarrow \exists\, t \in \mathit{ITEParcelExpr}(Pipe).$$
$$\text{`}v_k := t\text{'} \in C.\mathit{Tr} \wedge (e, b) \in \mathit{support}\, t$$

Datapath transformations

$$\forall\, dp \in \mathit{Dps}.$$
$$\forall\, pclP \in \mathit{Arg}(dp.\mathit{PclP}).\ \forall\, pclN \in \mathit{Arg}(dp.\mathit{PclN}).$$
$$(pclP, \mathbf{true}, pclN) \in FanOutEdges$$

4.3 Parcel Steps In *DiffAddMult*

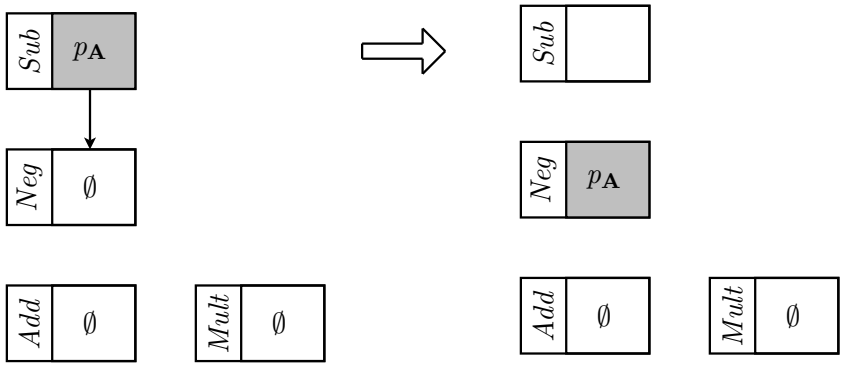


Figure 4.11: Pipeline step $(q_P^0, t_P^0, q_P^1) \in R_P$.

In this section we recall the *DiffAddMult* example introduced in Figure 3.1 from Section 3.1. *DiffAddMult* uses valid bits to represent whether data held by a parcel register is valid. Invalid data does not move through the pipeline. In the first state of the run, the parcel registers contain invalid data:

$$q_P^0 \models (valid_{Neg} = \mathbf{false} \wedge valid_{Add} = \mathbf{false} \wedge valid_{Mult1} = \mathbf{false} \wedge valid_{Mult2} = \mathbf{false})$$

As described in Section 3.1, in the *DiffAddMult* pipeline invalid register values do not generate requests and therefore do not propagate.

We consider a computation of the *DiffAddMult* pipeline, the first step of which is described in Figure 4.11. In this step the pipeline receives an input value that represents parcel $p_{\mathbf{A}}$ such that $p_{\mathbf{A}} = \{ v_i \}$. The parcel propagates through the Sub instance, the result of which is stored in register r_{Neg}. The Sub datapath produces the control output $selN_{Sub} = 001$.

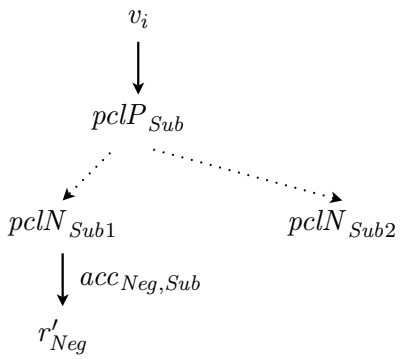


Fig. 4.12a. Fan-out graph.

Variable	Value
v_i	$\langle -30, 10, 5, \texttt{mult} \rangle$
$pclP_{Sub}$	$\langle -30, 10, 5, \texttt{mult} \rangle$
$pclN_{Sub1}$	$\langle -40, 5, \texttt{mult} \rangle$
$pclN_{Sub2}$	$\langle -40, 5 \rangle$
${r_{Neg}}'$	$\langle -40, 5, \texttt{mult} \rangle$
$selN_{Sub}$	*001*

Fig. 4.12b. Parcel and control environments.

Figure 4.12: Parcel $p_{\mathbf{A}} = \{ v_i \}$ in pipeline step (q_P^0, t_P^0, q_P^1).

The fan-out graph of parcel $p_{\mathbf{A}}$ and the parcel and control environments are described in Figure 4.12. The parcel corresponds to the operation $| - 30 - 10| \times 5$. It should therefore take the following path through the pipeline:

$$Sub \rightarrow Neg \rightarrow Mult \rightarrow Mult \ldots$$

The fan-out graph shows that the parcel propagates from the input variable v_i, through the Sub datapath and into the register r_{Neg}. There are two environments that describe the parcel step. The parcel environment valuates the variables in the fan-out of the parcel and expresses the datapath transformation of the parcel's values. The control environment valuates the control variables that appear as arguments to the datapaths that process the parcel. In this case, the transformation through the Sub datapath assigns *001* to the control variable $selN_{Sub}$.

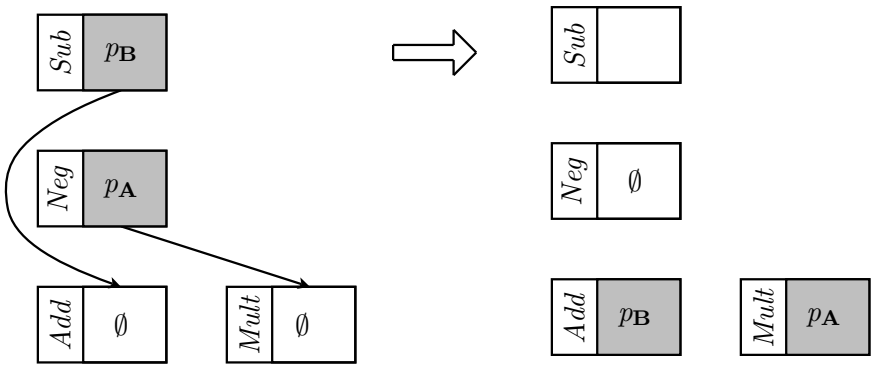


Figure 4.13: Pipeline step $(q_P^1, t_P^1, q_P^2) \in R_P$.

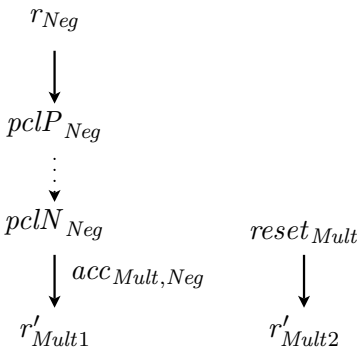


Fig. 4.14a. Fan-out graph.

Variable	Value
r_{Neg}	$\langle -40, 5, \texttt{mult} \rangle$
$pclP_{Neg}$	$\langle -40, 5, \texttt{mult} \rangle$
$pclN_{Neg}$	$\langle 40, 5 \rangle$
${r_{Mult1}}'$	$\langle 40, 5 \rangle$
${r_{Mult2}}'$	***
$selN_{Neg}$	*10*

Fig. 4.14b. Parcel and control environments.

Figure 4.14: Parcel $p_{\mathbf{A}} = \{ r_{Neg}, {r_{Mult2}}' \}$ in pipeline step (q_P^1, t_P^1, q_P^2).

In the second step, described in Figure 4.13, the pipeline transfers parcel $p_{\mathbf{A}}$ from r_{Neg} to r_{Mult1} and inputs a new parcel $p_{\mathbf{B}}$ into register r_{Add}. The two parcels and the control outputs they generate are described in Figure 4.14 and Figure 4.15. The newly received parcel $p_{\mathbf{B}}$ represents the operation $|12 - 10| + 2$ and takes the path $Sub \rightarrow Add$. In the current step the register r_{Mult2} is reset. Since this value is not actually used in the next cycle we display it as ***. Also, note that because the value assigned to r_{Mult2} is not dependent on the parcel's current values, to reflect its inclusion into $p_{\mathbf{A}}$ in the next step, we add ${r_{Mult2}}'$ to the parcel in the current step.

In the next step, the pipeline computation progresses as described in Figure 4.16. Parcel $p_{\mathbf{C}}$ enters the pipeline and propagates through the Sub datapath into r_{Neg}, as described in Figure 4.17. Parcel $p_{\mathbf{C}}$ denotes the operation $|2 - 3| + 5$. It therefore takes the path $Sub \rightarrow Neg \rightarrow Add$. Parcel $p_{\mathbf{A}}$

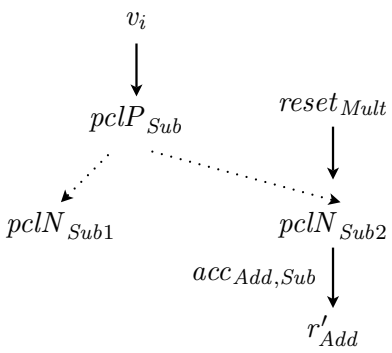


Fig. 4.15a. Fan-out graph.

Variable	Value
v_i	$\langle 12, 10, 2, \texttt{add} \rangle$
$pclP_{Sub}$	$\langle 12, 10, 2, \texttt{add} \rangle$
$pclN_{Sub1}$	$\langle 2, 2, \texttt{add} \rangle$
$pclN_{Sub2}$	$\langle 2, 2 \rangle$
$r_{Add}{}'$	$\langle 2, 2 \rangle$
$selN_{Sub}$	*010*

Fig. 4.15b. Parcel and control environments.

Figure 4.15: Parcel $p_{\mathbf{B}} = \{ v_i \}$ in pipeline step (q_P^1, t_P^1, q_P^2).

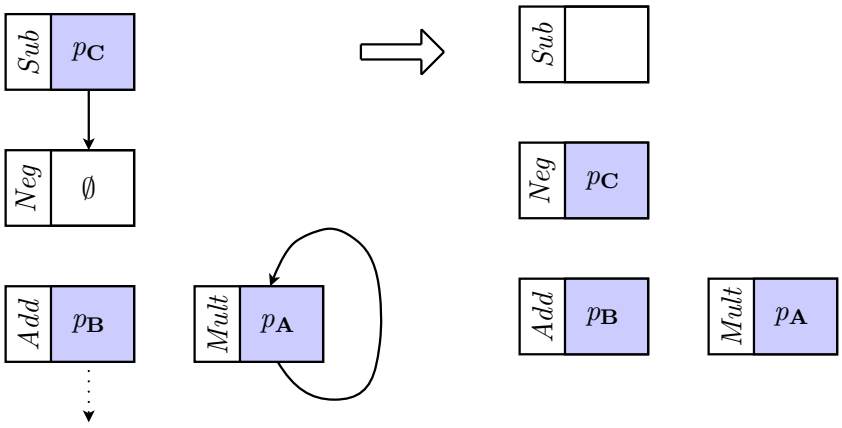


Figure 4.16: Pipeline step $(q_P^2, t_P^2, q_P^3) \in R_P$.

proceeds through one round of the multiplication operation. Parcel $p_{\mathbf{B}}$ undergoes the addition and is ready to tranfer out. To preserve ordering between $p_{\mathbf{A}}$ and $p_{\mathbf{B}}$, the pipeline control stalls parcel $p_{\mathbf{B}}$. The computation steps of the two parcels are represented in Figure 4.18 and Figure 4.19. The algorithm described in Section 3.1 performs the 8-bit multiplication by 4-bit multiplications and additions. For parcel $p_{\mathbf{A}}$, using the notation in Figure 3.8, we have $a \times c = \mathit{low}(40) \times \mathit{low}(5) = 8 \times 5$. The partial results of the multiplication are stored in r_{Mult2}. The fan-out graph of parcel $p_{\mathbf{A}}$ shows that the multiplication uses registers r_{Mult1} and r_{Mult2} as arguments, and in the next state, the value of register r_{Mult1} remains unchanged while r_{Mult2} gets updated with a new value produced by the multiplier. The parcel environment corresponding to $p_{\mathbf{A}}$ valuates all variables in the parcel's fan-out graph. We do not show values that are inconsistent at this stage of the multiplication, and

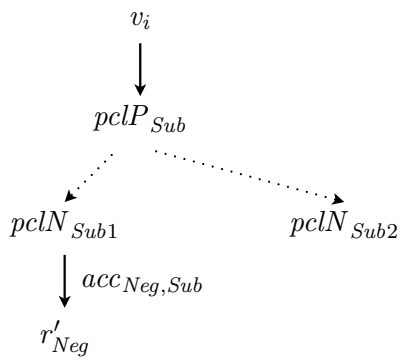


Fig. 4.17a. Fan-out graph.

Variable	Value
v_i	$\langle 2, 3, 5, \texttt{add} \rangle$
$pclP_{Sub}$	$\langle 2, 3, 5, \texttt{add} \rangle$
$pclN_{Sub1}$	$\langle -1, 5, \texttt{add} \rangle$
$pclN_{Sub2}$	$\langle -1, 5 \rangle$
$r_{Neg}{}'$	$\langle -1, 5, \texttt{add} \rangle$
$selN_{Sub}$	*001*

Fig. 4.17b. Parcel and control environments.

Figure 4.17: Parcel $p_{\mathbf{C}} = \{ v_i \}$ in pipeline step (q_P^2, t_P^2, q_P^3).

denote them by ***. The control environment shows the current and the next control states of the multiplication.

The fourth step of the pipeline computation is shown in Figure 4.20. In this step the Sub datapath processes the newly input parcel $p_{\mathbf{D}}$. Parcel $p_{\mathbf{D}}$ is a multiplication operation and its request to the multiplier datapath is not accepted. Parcel $p_{\mathbf{B}}$ is once again stalled to preserve ordering with parcel $p_{\mathbf{A}}$ which progresses through another step of the multiplication operation, as described in Figure 4.22. Because of $p_{\mathbf{C}}$, $p_{\mathbf{B}}$ also stalls, as shown in Figure 4.21.

In the fifth step of the pipeline computation, described in Figure 4.23, parcel $p_{\mathbf{A}}$ completes the multiplication, as shown in Figure 4.25, and therefore, both it and parcel $p_{\mathbf{B}}$ transfer out of the pipeline. The progress of parcel $p_{\mathbf{C}}$ is described in Figure 4.24. Since the $Mult$ datapath is free, parcel $p_{\mathbf{D}}$ is granted its request and moves into r_{Mult1}.

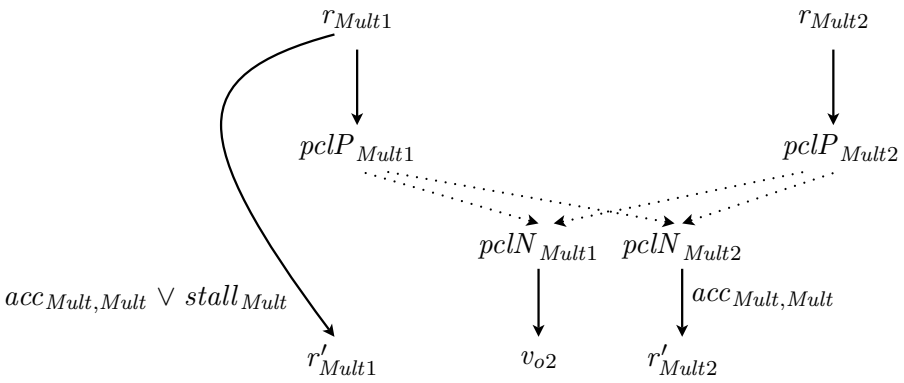


Fig. 4.18a. Fan-out graph.

Variable	Value
r_{Mult1}	$\langle 40, 5 \rangle$
r_{Mult2}	***
$pclP_{Mult1}$	$\langle 40, 5 \rangle$
$pclP_{Mult2}$	r_{Mult2}
$pclN_{Mult1}$	***
$pclN_{Mult2}$	$\langle a \times c = 8 \times 5 \rangle$
${r_{Mult1}}'$	$\langle 40, 5 \rangle$
${r_{Mult2}}'$	$\langle a \times c = 8 \times 5 \rangle$
v_{o2}	***
$stateIn$	*000*
$stateOut$	*010*

Fig. 4.18b. Parcel and control environments.

Figure 4.18: Parcel $p_{\mathbf{A}} = \{ r_{Mult1}, r_{Mult2} \}$ in pipeline step (q_P^2, t_P^2, q_P^3).

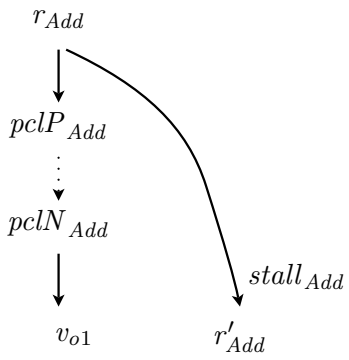


Fig. 4.19a. Fan-out graph.

Variable	Value
r_{Add}	$\langle 2, 2 \rangle$
$pclP_{Add}$	$\langle 2, 2 \rangle$
$pclN_{Add}$	$\langle 4 \rangle$
v_{o1}	$\langle 4 \rangle$
$r_{Add}{}'$	$\langle 2, 2 \rangle$

Fig. 4.19b. Parcel and control environments.

Figure 4.19: Parcel $p_{\mathbf{B}} = \{\ r_{Add}\ \}$ in pipeline step (q_P^2, t_P^2, q_P^3).

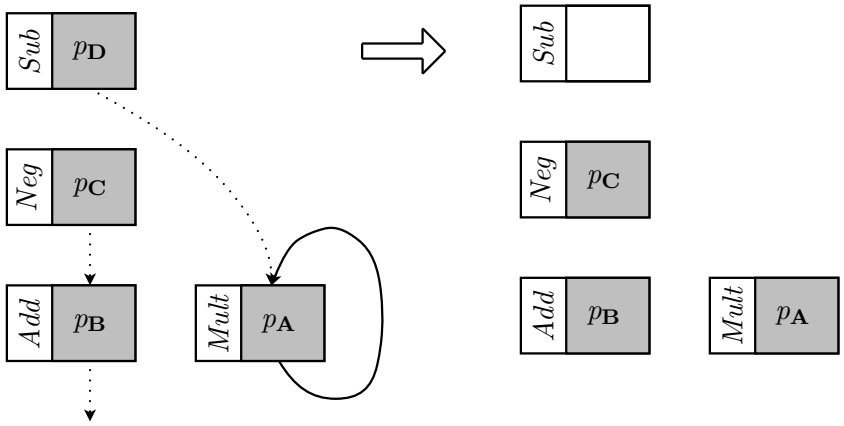
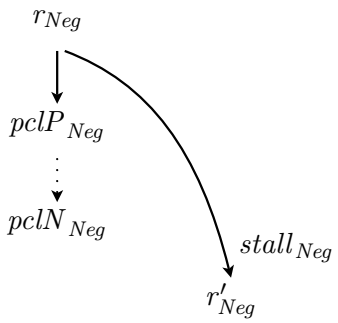


Figure 4.20: Pipeline step $(q_P^3, t_P^3, q_P^4) \in R_P$.



Variable	Value
r_{Neg}	$\langle -1, 5, \texttt{add} \rangle$
$pclP_{Neg}$	$\langle -1, 5, \texttt{add} \rangle$
$pclN_{Neg}$	$\langle 1, 5 \rangle$
${r_{Neg}}'$	$\langle -1, 5, \texttt{add} \rangle$
$selN_{Neg}$	*01*

Fig. 4.21a. Fan-out graph.

Fig. 4.21b. Parcel and control environments.

Figure 4.21: Parcel $p_{\mathbf{C}} = \{\, r_{Neg} \,\}$ in pipeline step (q_P^3, t_P^3, q_P^4).

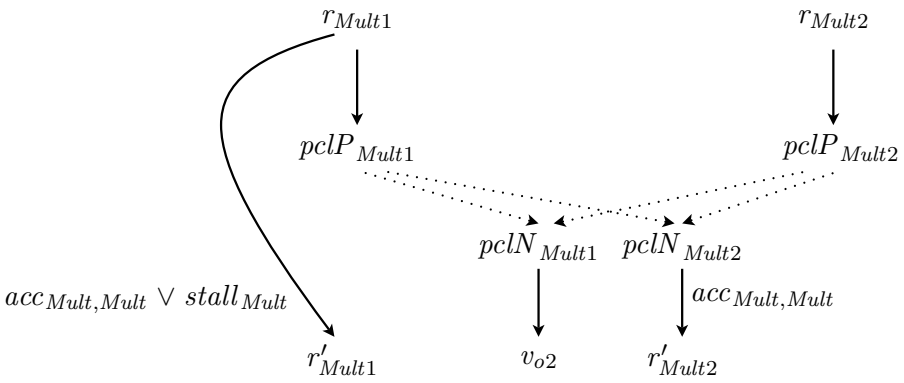


Fig. 4.22a. Fan-out graph.

Variable	Value
r_{Mult1}	$\langle 40, 5 \rangle$
r_{Mult2}	$\langle ac = 8 \times 5 \rangle$
$pclP_{Mult1}$	$\langle 40, 5 \rangle$
$pclP_{Mult2}$	r_{Mult2}
$pclN_{Mult1}$	***
$pclN_{Mult2}$	$\langle ac = 8 \times 5, \mathit{low}(bc) = 10 \rangle$
${r_{Mult1}}'$	$\langle 40, 5 \rangle$
${r_{Mult2}}'$	$\langle ac = 8 \times 5, \mathit{low}(bc) = 10 \rangle$
v_{o2}	***
$stateIn$	*010*
$stateOut$	*100*

Fig. 4.22b. Parcel and control environments.

Figure 4.22: Parcel $p_{\mathbf{A}} = \{ r_{Mult1}, r_{Mult2} \}$ in pipeline step (q_P^3, t_P^3, q_P^4).

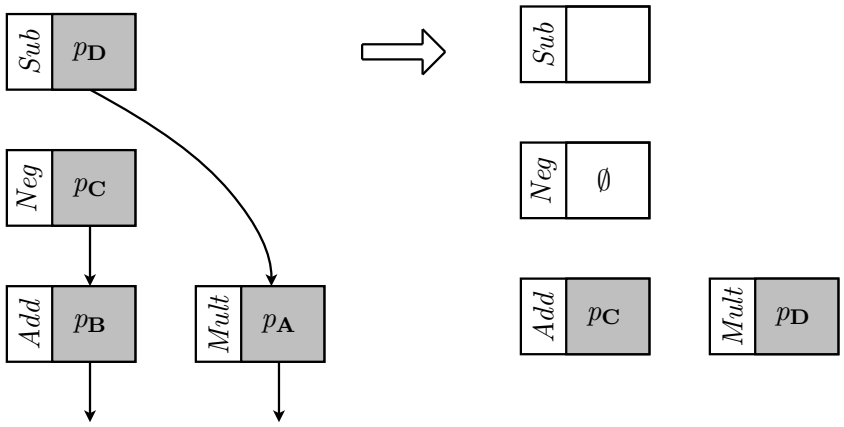


Figure 4.23: Pipeline step $(q_P^4, t_P^4, q_P^5) \in R_P$.

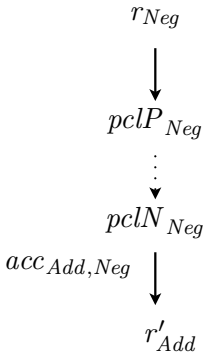


Fig. 4.24a. Fan-out graph.

Variable	Value
r_{Neg}	$\langle -1, 5, \texttt{add} \rangle$
$pclP_{Neg}$	$\langle -1, 5, \texttt{add} \rangle$
$pclN_{Neg}$	$\langle 1, 5 \rangle$
$r_{Add}{}'$	$\langle 1, 5 \rangle$
$selN_{Neg}$	*01*

Fig. 4.24b. Parcel and control environments.

Figure 4.24: Parcel $p_{\mathbf{C}} = \{\, r_{Neg} \,\}$ in pipeline step (q_P^4, t_P^4, q_P^5).

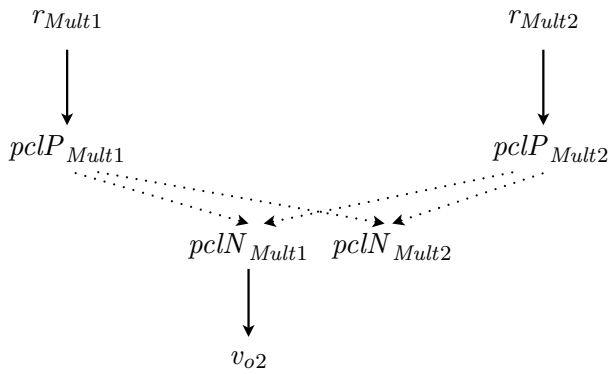


Fig. 4.25a. Fan-out graph.

Variable	Value
r_{Mult1}	$\langle 40, 5 \rangle$
r_{Mult2}	$\langle ac = 8 \times 5, \mathit{low}(bc) = 10 \rangle$
$pclP_{Mult1}$	$\langle 40, 5 \rangle$
$pclP_{Mult2}$	r_{Mult2}
$pclN_{Mult1}$	$\langle 200 \rangle$
$pclN_{Mult2}$	***
v_{o2}	$\langle 200 \rangle$
$stateIn$	*100*
$stateOut$	*111*

Fig. 4.25b. Parcel and control environments.

Figure 4.25: Parcel $p_{\mathbf{A}} = \{ r_{Mult1}, r_{Mult2} \}$ in pipeline step (q_P^4, t_P^4, q_P^5).

Our example illustrates the concept of parcel step. The parcel is a group of parcel variables, register, combinational and next-state — in the case when the parcel incorporates a register variable that is being assigned a constant or **choice**. The parcel step is a record of the parcel's transformation in one pipeline step. The fan-out graph of the parcel shows the propagation of the parcel's values and the two environments, parcel and control, document the computations of the datapaths.

4.4 Parcel Steps

A parcel computation consists of a sequence of parcel steps, each such step occurring within a corresponding pipeline step. A parcel step consists of the parcel's current state, transition label and next state.

The state of a parcel is a substate of the pipeline model. It is defined as an environment over parcel registers. The state of p is $q_{PA} \in \mathit{PEnv}(\mathit{RegPcl})$ such that:

$$q_{PA} = q_P \mid_{p \cap \mathit{RegPcl}}$$

Since the domain of q_P does not contain combinational variables, we have $q_P \mid_{p \cap \mathit{RegPcl}} = q_P \mid_{p}$ and therefore we can write

$$q_{PA} = q_P \mid_{p}$$

If a parcel contains no registers then its state is the empty environment $\emptyset$.

4.4.1 Definition (Parcel Step)**.** A parcel step of parcel $p \subseteq V_{pcl} \cup \mathit{NextRegPcl}$ is a triplet $(q_{PA}, t_{PA}, {q_{PA}}')$:

- $q_{PA} \in \mathit{PEnv}(\mathit{RegPcl})$ is the parcel's current state.
- ${q_{PA}}' \in \mathit{PEnv}(\mathit{RegPcl})$ is the parcel's next state.
- $t_{PA} = \langle\langle Nodes, Succ\rangle, e_{pcl}, e_{ctrl}\rangle$ is the step label.
- $\langle Nodes, Succ\rangle$ is a fan-out graph.
- $\mathit{roots}\ \langle Nodes, Succ\rangle = p$
- $e_{pcl} \in \mathit{Env}(Nodes \cap \mathit{CombPcl})$
- $e_{ctrl} \in \mathit{PEnv}(V_{ctrl})$
- $\mathit{dom}\ q_{PA} = Nodes \cap \mathit{RegPcl}$
- $\mathit{dom}\ {q_{PA}}' = \{\, v \mid v' \in Nodes \cap \mathit{NextRegPcl} \,\}$

- $Nodes$ contains all the parcel arguments of the referenced datapaths.

$$\bigcup_{dp \in \textit{datapaths}\ \langle Nodes, Succ\rangle} \textit{Arg}(dp.V_{pcl}) \subseteq Nodes \tag{4.1}$$

- e_{ctrl} valuates exactly the control input and output arguments of the referenced datapaths:

$$\textit{dom}(e_{ctrl}) = \bigcup_{dp \in \textit{datapaths}\ \langle Nodes, Succ\rangle} \textit{Arg}(dp.V_{ctrl}) \tag{4.2}$$

A parcel step corresponds to the propagation and transformation of the parcel's values through the variables and datapath instances in its fan-out. Propagation consists of copying through variables and datapath transformations. The transition label of the parcel step describes the parcel between the two endpoints, its current and next state and captures the behaviour of the datapaths that transform the parcel. The label consists of a tuple $\langle \textit{fanOut}\ p, e_{pcl}, e_{ctrl}\rangle$ where e_{pcl} and e_{ctrl} are parcel, and respectively, control environments over the variables in the fan out of the parcel. The transition label is a full record of the parcel's effect on combinational variables.

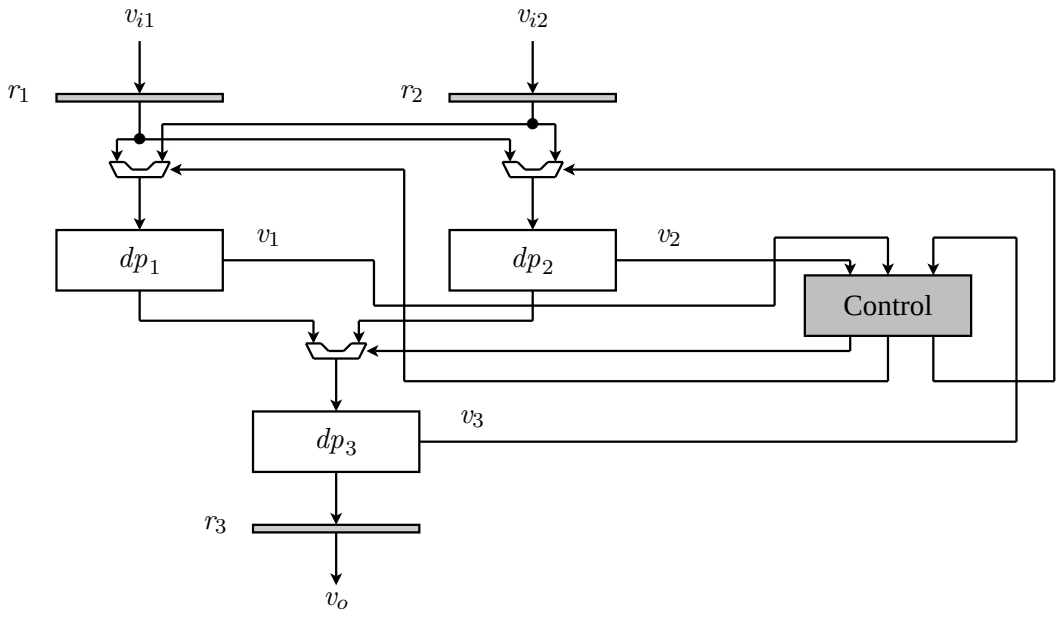


Figure 4.26: Parcel $p = \{ r_1, r_2 \}$ can have up to 8 distinct fan-out graphs.

We use the pipeline model in Figure 4.26 to illustrate the need to have the parcel's fan-out graph on the transition label. The parcel $p = \{ r_1, r_2 \}$ can propagate into register r_3 in $2 \times 2 \times 2 = 8$ different ways, given that for each datapath there are two different ways to select its input argument. At a minimum, the label of the parcel's transition should describe the arguments to each datapath in

terms of values derived transitively from the parcel's values. In our example, the conditions on the edges of the fan-out graph of the parcel, i.e. the mux select signals, may depend on control variables that are not derived from the parcel.

In a pipeline step, the parcel's values move through combinational datapaths and parcel variables, the results of which propagate into next state parcel registers, which in turn, denote the next state of the parcel parcel. For a parcel p the parcel step contained in the pipeline model step is a triplet $(q_{PA}, t_{PA}, q_{PA}{}')$ such that:

- q_{PA} is the parcel's current state.
- t_{PA} is the step label.
- $q_{PA}{}'$ is the parcel's next state:

$$q_{PA}{}' = q_P{}' \mid_{\{\, v \mid v' \in p^* \cap \mathit{NextRegPcl} \,\}}$$

- $t_{PA} = \langle \mathit{fanOut}\ p, e_{pcl}, e_{ctrl} \rangle$ where:
  - $e_{pcl} \in \mathit{Env}(p^* \cap \mathit{CombPcl})$ is defined by:

$$e_{pcl} = t_P \mid_{p^* \cap \mathit{CombPcl}} = t_P \mid_{\mathit{CombPcl}}$$

  - $e_{ctrl} \in \mathit{PEnv}(V_{ctrl})$ where:

$$\begin{aligned} \mathit{dom}\ e_{ctrl} &= \bigcup_{dp \in \mathit{datapaths}\ (\mathit{fanOut}\ p)} \mathit{Arg}(dp.V_{ctrl}) \\ e_{ctrl}(v) &= (q_P \cup t_P)(v) \end{aligned}$$

The parcel's next state is obtained by projecting out the pipeline's next state over the next-state registers in the parcel's fan-out. The parcel environment is obtained by projecting out the pipeline transition label with respect to the combinational variables in the parcel's fan-out. The control environment is defined over the actual parameters for input and output control variables of the datapaths referenced by the fan-out graph. For each variable in its domain, the control environment returns its value in the pipeline step.

Given a parcel p, we denote its next state by *pclNextState* p. The transition label of the corresponding parcel step is denoted by *pclTrans* p. The step itself is denoted by *pclStep* p.

As an example consider $p_{\mathbf{C}} = \{\, r_{Neg} \,\}$ in the pipeline step (q_P^4, t_P^4, q_P^5), shown in Figure 4.27. According to the definition we have:

$$q_{PA} = q_P^4 \mid_{r_{Neg}} = \langle -1, 5, \texttt{add} \rangle$$

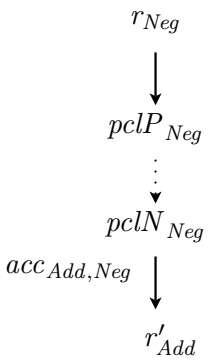


Fig. 4.27a. Fan-out graph.

Variable	Value
r_{Neg}	$\langle -1, 5, \texttt{add} \rangle$
$pclP_{Neg}$	$\langle -1, 5, \texttt{add} \rangle$
$pclN_{Neg}$	$\langle 1, 5 \rangle$
${r_{Add}}'$	$\langle 1, 5 \rangle$
$selN_{Neg}$	*01*

Fig. 4.27b. Parcel and control environments.

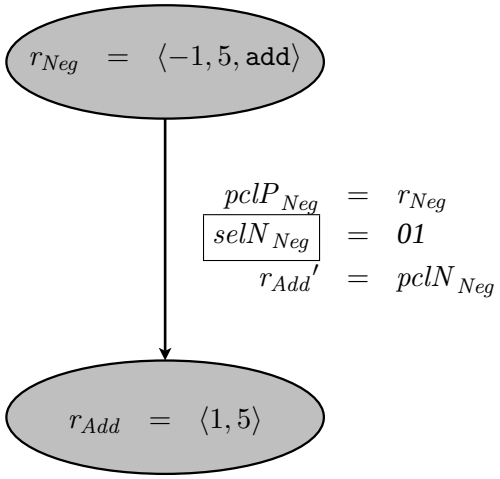


Fig. 4.27c. Parcel Step

Figure 4.27: Parcel step for parcel $p_{\mathbf{C}}$.

$$
\begin{aligned}
t_{PA} &= \langle \textit{fanOut}\ p, e_{pcl}, e_{ctrl} \rangle \\
{q_{PA}}' &= q_P^5 \mid_{r_{Add}} = \langle 1, 5 \rangle \\
e_{pcl}(pclP_{Neg}) &= \langle -1, 5, \texttt{add} \rangle \\
e_{pcl}(pclN_{Neg}) &= \langle 1, 5 \rangle \\
e_{ctrl}(selN_{Neg}) &= \textit{01}
\end{aligned}
$$

The parcel step for $p_{\mathbf{C}}$ is described graphically in Figure 4.27c. The transition label is described by a set of assignments that can be used to infer the fan-out graph, the parcel and the control environments.

The natural generalization of a parcel step contained in a pipeline step is to remove the context of the pipeline step (q_P, t_P, q'_P) and use standalone environments e_{pcl} and e_{ctrl}. The parcel environment

$(q_{PA} \cup e_{pcl} \cup {q_{PA}}' \left[{}^{NextRegPcl}/_{RegPcl}\right])$ must implement value copying correctly:

$$\begin{aligned} &\forall\, (v_l, b, v_k) \in (\mathit{fanOut}\ p).\mathit{Succ}.\ v_l \notin \mathit{PclP} \\ &\quad\Longrightarrow \\ &\quad (q_{PA} \cup e_{pcl} \cup {q_{PA}}' \left[{}^{NextRegPcl}/_{RegPcl}\right])(v_k) = (q_{PA} \cup e_{pcl} \cup {q_{PA}}' \left[{}^{NextRegPcl}/_{RegPcl}\right]) \end{aligned} \tag{4.3}$$

4.5 Parcel Automata

A parcel automaton is a labeled transition system defined by parcel steps. The transitions of the parcel automaton have form $(q_{PA}, t_{PA}, {q_{PA}}')$ for a parcel step $(q_{PA}, t_{PA}, {q_{PA}}'')$ such that ${q_{PA}}' \subseteq {q_{PA}}''$.

4.5.1 Definition (Parcel Automaton)**.** A parcel automaton is a labeled transition system $\langle Q_{PA}, R_{PA}, T_{PA}, I_{PA}\rangle$ such that:

- $Q_{PA} \subseteq \mathit{PEnv}(\mathit{RegPcl}) \cup \{\, \mathit{final}_{PA} \,\}$.
- T_{PA} consists of a subset of parcel step labels and the empty transition label $\emptyset$.
- $R_{PA} \subseteq Q_{PA} \times T_{PA} \times Q_{PA}$ consists of a set of parcel transitions:

  $$\{\, (q_{PA}, t_{PA}, {q_{PA}}') \mid {q_{PA}}' \neq \emptyset \wedge \exists\, {q_{PA}}''.\ {q_{PA}}' \subseteq {q_{PA}}'' \wedge (q_{PA}, t_{PA}, {q_{PA}}'') \text{ is a parcel step } \} \cup$$
  $$\{\, (q_{PA}, t_{PA}, \mathit{final}_{PA}) \mid \exists\, {q_{PA}}'.\ (q_{PA}, t_{PA}, {q_{PA}}') \text{ is a parcel step } \}$$

- $(\mathit{final}_{PA}, \emptyset, \mathit{final}_{PA}) \in R_{PA}$
- $I_{PA} \subseteq Q_{PA}$.

We denote the set of parcel automata for a given pipeline model $Pipe$ by $\mathit{Pa}(Pipe)$.

The parcel automaton described in Figure 4.28 and Figure 4.29 models the parcel computation for parcel $p_{\mathbf{A}}$ in our example. In the first step, we have $p_{\mathbf{A}} = \{\, v_i \,\}$. Since the parcel consists of a combinational variable, its state is $\emptyset$. By reading the label on the outgoing transition from state $\emptyset$ we infer that the parcel propagates through the Sub datapath into register r_{Neg}. The datapath control inputs and outputs are highlighted on the transition label. The Sub datapath has the control output $selN_{Sub} = \mathit{001}$. The second transition describes the movement of the parcel through the Neg datapath into r_{Mult1}. The assignment ${r_{Mult2}}' := reset_{Mult}$ assigns $\langle 0, 0\rangle$ that denotes an 8-bit number and a 4-bit one. The remaining two transitions of the parcel automaton conclude the parcel computation. In the last transition, the automaton reaches the final state denoted by final_{PA}, from which only the empty transition $\emptyset$ is possible.

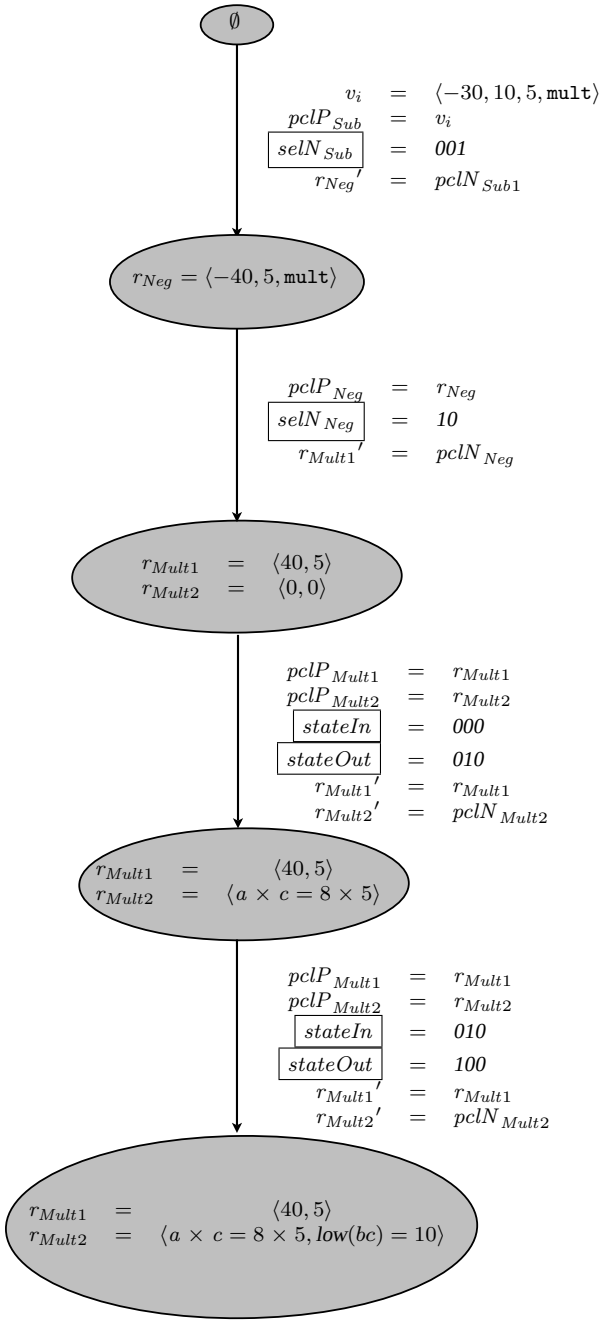


Figure 4.28: Parcel automaton for parcel $p_{\mathbf{A}}$.

Figure 4.30 describes the transition of a parcel automaton that models the stall of parcel $p_{\mathbf{C}}$ in step (q_P^3, t_P^3, q_P^4), previously described in Figure 4.21. During a stall, the state of the parcel automaton does not change.

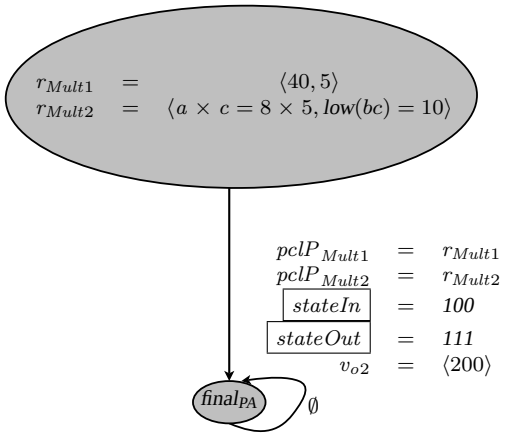


Figure 4.29: Parcel automaton for parcel $p_{\mathbf{A}}$ (continuation).

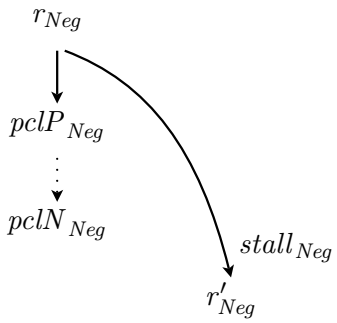


Fig. 4.30a. Fan-out graph.

Variable	Value
r_{Neg}	$\langle -1, 5, \texttt{add}\rangle$
$pclP_{Neg}$	$\langle -1, 5, \texttt{add}\rangle$
$pclN_{Neg}$	$\langle 1, 5\rangle$
$r_{Neg}{}'$	$\langle -1, 5, \texttt{add}\rangle$
$selN_{Neg}$	*01*

Fig. 4.30b. Parcel and control environments.

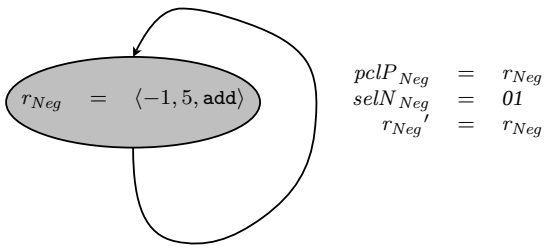


Fig. 4.30c. Parcel transition for a stall.

Figure 4.30: Parcel $p_{\mathbf{C}} = \{\, r_{Neg} \,\}$ in pipeline step (q_P^3, t_P^3, q_P^4).

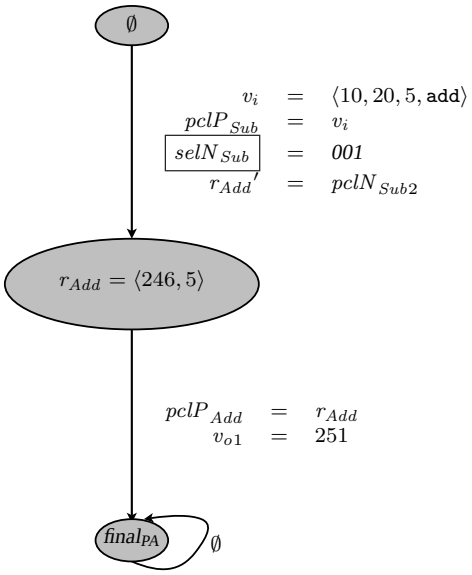


Figure 4.31: Parcel automaton illustrating an unreachable parcel computation.

Parcel automata can also describe computations that do not correspond to actual parcel computations in the pipeline model. In parcel computations embedded in pipeline computations control dependencies between control variables are determined by assignments. In the computations of a parcel automaton, control variables are free and such dependencies may not be preserved. The parcel automaton in Figure 4.31 describes a parcel computation that is not reachable in *DiffAddMult*. The condition on the fan-out edge $(pclN_{Sub}, acc_{Add,Sub}, r_{Add}{}')$ is not true when $selN_{Sub} = 001$. In the parcel automaton the variable $acc_{Add,Sub}$ appears to be independent of $selN_{Sub}$. The addition operation interprets the tuple $\langle -10, 5\rangle$ as a pair of two positive 8-bit numbers. In two's complement -10 is represented as $256 - 10$ which leads to the addition producing the result $246 + 5$.

We can characterize parcel steps $(q_{PA}, \langle \mathit{fanOut}\ p, e_{pcl}, e_{ctrl}\rangle, q_{PA}{}')$ that are consistent with the behaviour of the pipeline datapath. The inputs and output arguments of each datapath instance that appears in a parcel step must be transformed according to a step of the corresponding datapath. For each datapath instance there exists a step such that the input and output variables on its transition label are the same as the arguments provided by the environment $e_{pcl} \cup e_{ctrl}$.

$$\begin{aligned} &\forall\ dp \in \mathit{datapaths}\ (\mathit{fanOut}\ p).\\ &\quad \exists\ (q_D, t_D, q_D') \in \mathit{LTS}(dp.C).R_C.\\ &\qquad \forall\ v \in dp.V_{pcl} \cup dp.V_{ctrl}.\\ &\qquad\quad (e_{ctrl} \cup e_{pcl})(\mathit{Arg}(v)) = t_D(v) \end{aligned} \tag{4.4}$$

When Equation 4.4 holds, we say the transition satisfies value propagation through the datapath.

4.5.2 Definition. Consistent Parcel Automaton We call a parcel automaton *consistent* if its transitions satisfy value propagation through the pipeline datapaths.

Inconsistent parcel automata are also possible. For instance, the parcel automaton in Figure 4.32 is perfectly legal even though the subtraction produces the incorrect result $\langle 4, 2\rangle$ instead of $\langle 5, 2\rangle$.

4.6 Abstractions Of Parcel Automata

Datapath abstraction replaces the concrete datapaths with abstract ones that retain the control visible behaviour of the datapath. The partial orders on parcel automata such as simulation and language containment provide our definition of control visible datapath behaviour.

When comparing concrete values to abstract values we must provide the same context. The context is given by the registers that hold the parcel's values and the fan-out graph that specifies how the values propagate. We define the label of parcel automaton states and transitions so that the contexts of the values and behaviours they represent are the same. Since the control visible datapath

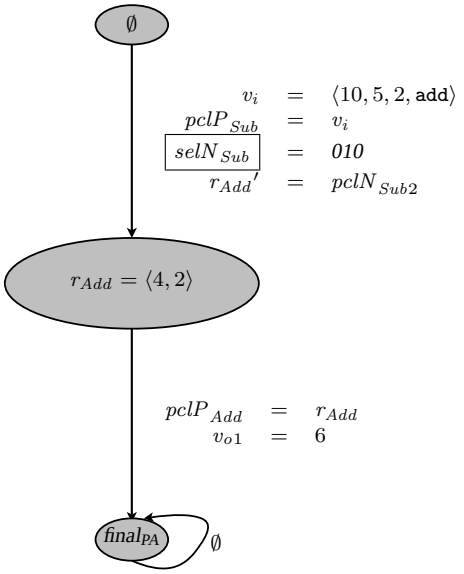


Figure 4.32: Parcel automaton showing inconsistent datapath behaviour.

behaviour that we are abstracting for appears on the edges of the parcel automaton, the label of the parcel automaton edge must also contain the values of the control variables that appear on the edges the fan-out of the parcel.

In the remainder of the thesis we use the following notation for the concrete and abstract parcel automata:

$$\begin{aligned} pa_c &= \langle Q_{PAc}, R_{PAc}, T_{PAc}, I_{PAc} \rangle \\ pa_a &= \langle Q_{PAa}, R_{PAa}, T_{PAa}, I_{PAa} \rangle \end{aligned}$$

Therefore, parcel states have the same label if they are defined over the same set of register variables. Transitions have the same label if their fan-out graphs and control environments coincide. According to the definition of abstract interpretation of pipeline models presented in Section 3.3, the ITE expressions e_c and respectively, e_a that are assigned to a parcel variable v satisfy the condition $e_c \approx_{ai} e_a$: occurrences of concrete constants in e_c may be replaced by abstract constants in e_a. We employ '$\approx_{ai}$' to define equivalence of fan-out graphs.

We use the notation

$$\begin{aligned} t_{PAc} &= \langle fg_c, e_{pclc}, e_{ctrlc} \rangle \\ t_{PAa} &= \langle fg_a, e_{pcla}, e_{ctrla} \rangle \end{aligned}$$

We denote label equality for both states and transitions using the operator '$=_{PA}$':

$$q_{PAc} =_{PA} q_{PAa} \equiv (\mathit{dom}\, q_{PAc} = \mathit{dom}\, q_{PAa}) \vee (q_{PAc} = \mathit{final}_{PAc} \wedge q_{PAa} = \mathit{final}_{PAa}) \tag{4.5}$$

$$t_{PAc} =_{PA} t_{PAa} \equiv \left(\begin{array}{c} \left(\begin{array}{c} \forall\, (e_c, b, v) \in fg_c.\mathit{Succ}.\ \exists\, (e_a, b, v) \in fg_c.\mathit{Succ}.\ e_c \approx_{ai} e_a \\ \wedge \\ \forall\, (e_a, b, v) \in fg_c.\mathit{Succ}.\ \exists\, (e_c, b, v) \in fg_c.\mathit{Succ}.\ e_a \approx_{ai} e_c \end{array} \right) \\ \wedge \\ e_{ctrlc} = e_{ctrla} \\ \vee \\ t_{PAc} = \emptyset \wedge t_{PAa} = \emptyset \end{array} \right) \tag{4.6}$$

In Equation 4.6 the comparison of the fan-out graphs is made modulo '$\approx_{ai}$': the fan-out graphs are identical with the exception of constants. In that equation e_c and e_a may only be constants or variables.

We say an abstract parcel automaton is consistent with respect to constants if the abstract fan-out edges corresponding to a concrete fan-out edge (w_c, b, v) are all identical to (w_a, b, v). If the parcel automaton is consistent with respect to constants, then in abstract interpretation, the constant w_c is replaced by w_a.

Figure 4.33 and Figure 4.34 describe an abstract parcel automaton that represents parcel computations of an abstraction of the *DiffAddMult* pipeline model. The abstract model *DiffAddMult*$_a$ is defined as an abstract interpretation of the concrete one. We denote the abstract datapaths by Sub_a, Neg_a, Add_a, $Mult_a$. From the parcel automaton we infer that the datapath Sub_a is non-deterministic:

$pclP_{Sub}$	$selN_{Sub}$	$pclN_{Sub1}$	$pclN_{Sub2}$
α_0	*010*	α_2	α_2
α_0	*001*	α_1	α_1
α_0	*100*	α_4	α_4

Abstract multiplication is also non-deterministic. We use '***' to denote any constant in $\mathbf{B}^3$. When the parcel automaton is in the abstract state q_{PAa} defined by

$$\begin{aligned} q_{PAa}(r_{Mult1}) &= \alpha_6 \\ q_{PAa}(r_{Mult2}) &= \alpha_7 \end{aligned}$$

it can non-deterministically remain in the same state for an arbitrary number of transitions under any possible combination of $stateIn$ and $stateOut$ or it can transition to the final state, indicating

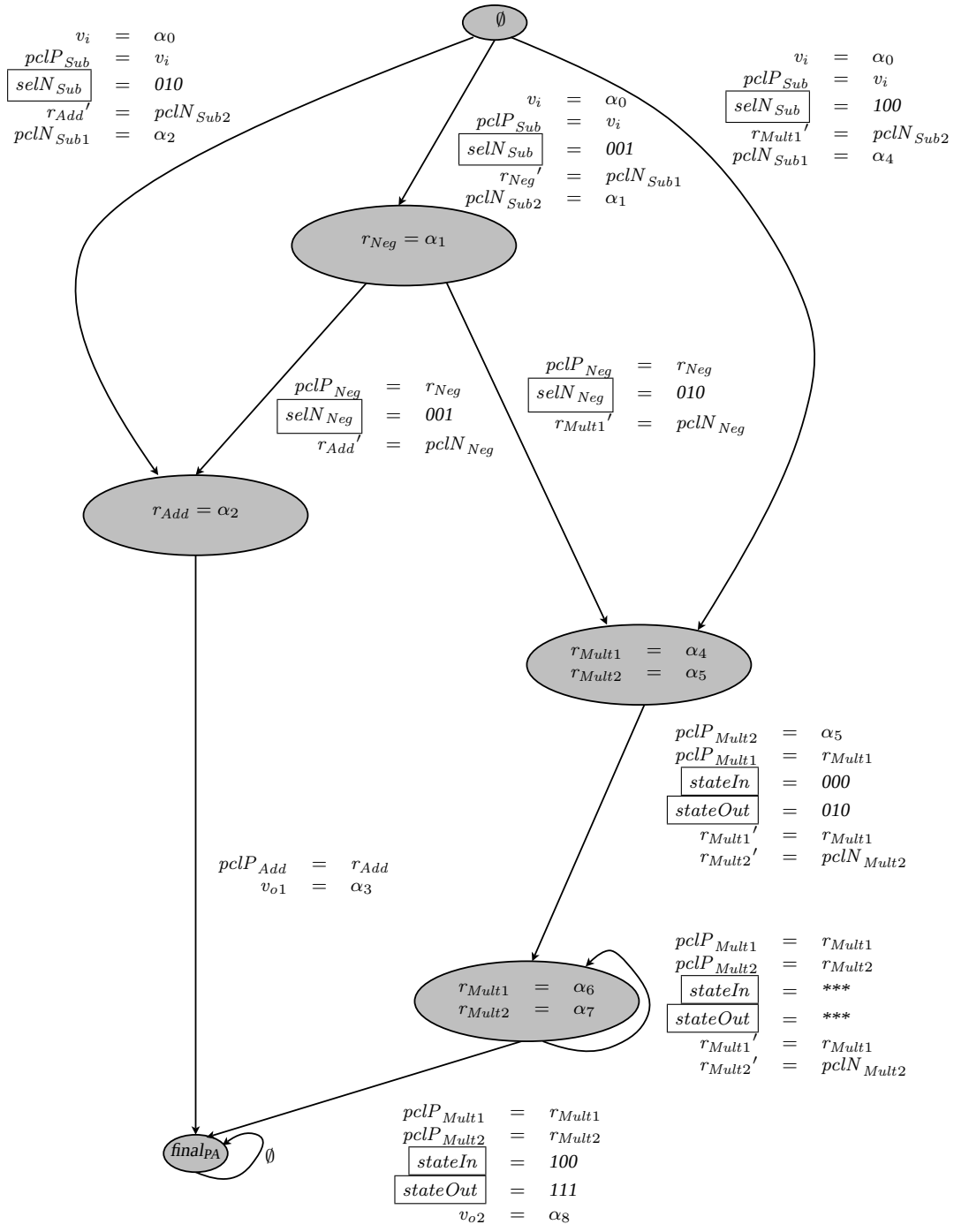


Figure 4.33: Abstract parcel automaton.

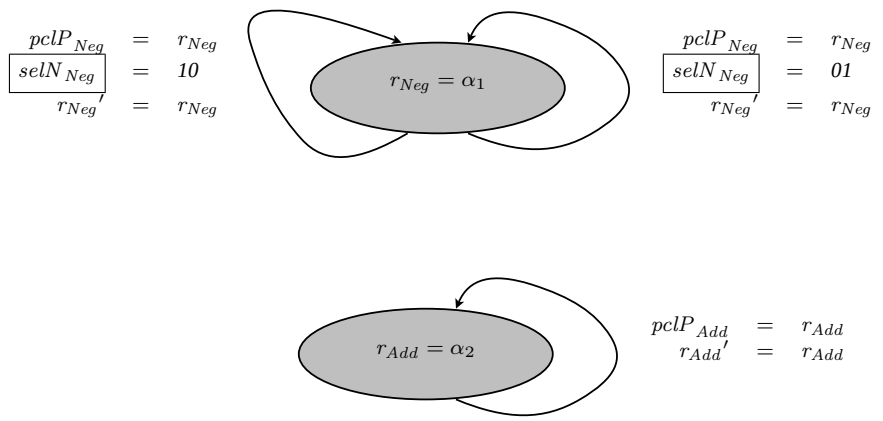


Figure 4.34: Abstract parcel automaton (continuation).

the end of the multiplication. Intuitively, the abstract parcel automaton has equivalent computations to each of the parcel computations described in Section 4.3.

Simulation on parcel automata is a simulation between labeled transition systems that preserves label equality on states and transitions.

4.6.1 Definition (Simulation Of Parcel Automata)**.** A simulation relation $\mathcal{S}_{PA} \subseteq Q_{PAc} \times Q_{PAa}$ between the two labeled transition systems is a parcel automata simulation relation if:

- $\mathcal{S}_{PA}$ respects the state labeling:

$$\forall\, q_{PAc}.\ \forall\, q_{PAa}.\ (q_{PAc}, q_{PAa}) \in \mathcal{S}_{PA} \Longrightarrow q_{PAc} =_{PA} q_{PAa} \tag{4.7}$$

- $\mathcal{S}_{PA}$ respects the transition labeling:

$$\begin{array}{ccc} q_{PAa} & \xrightarrow{t_{PAa}} & q_{PAa}' \\ \mathcal{S}_{PA} \uparrow & & \uparrow \mathcal{S}_{PA} \\ q_{PAc} & \xrightarrow{t_{PAc}} & q_{PAc}' \end{array} \Longrightarrow t_{PAc} =_{PA} t_{PAa} \tag{4.8}$$

- $\mathcal{S}_{PA}$ satisfies the property:

$$\begin{array}{l} \forall\, q_{PAc}.\ \forall\, q_{PAa}.\ (q_{PAc}, q_{PAa}) \in \mathcal{S}_{PA} \Longrightarrow \\ \quad \forall\, vars \subseteq \mathit{dom}\, q_{PAc}.\ (q_{PAc} \mid_{vars}, q_{PAa} \mid_{vars}) \in \mathcal{S}_{PA} \end{array} \tag{4.9}$$

Equation 4.9 in the definition of simulation of parcel automata is demanded by the fact that parcel automata states are composites. A transition $(q_{PA1}, t_{PA}, {q_{PA}}') \in R_{PA}$ corresponds to a parcel step which uses only the current state registers in q_{PA1}. It is therefore semantically correct to have the transition $(q_{PA2}, t_{PA}, {q_{PA}}')$ for any superstate $q_{PA1} \subseteq q_{PA2}$. It is also possible that given a substate $q_{PA2} \subseteq q_{PA1}$ the transition $(q_{PA2}, t_{PA}, {q_{PA}}')$ is still well defined. In both cases, such transitions, though allowed under the definition of the parcel step, might not be part of R_{PA}. This is exactly the reason for which Equation 4.9 compensates.

4.6.2 Definition. We call a parcel automaton *closed* if it satisfies the following two conditions:

- For each transition $(q_{PA1}, t_{PA}, {q_{PA}}') \in R_{PA}$, the transition relation R_{PA} contains all the similar well-defined transitions of both substates and superstates q_{PA2} of q_{PA1}:

$$\begin{aligned} &\forall\, (q_{PA1}, t_{PA}, {q_{PA}}') \in R_{PA}. \\ &\left(\begin{array}{l} \forall\, q_{PA2} \in Q_{PA}. \\ \quad q_{PA2} \subseteq q_{PA1} \wedge (q_{PA2}, t_{PA}, {q_{PA}}') \text{ is legal } \Longrightarrow (q_{PA2}, t_{PA}, {q_{PA}}') \in R_{PA}) \end{array} \right) \\ &\qquad\qquad \wedge \\ &\left(\begin{array}{l} \forall\, q_{PA2} \in Q_{PA}. \\ \quad q_{PA1} \subseteq q_{PA2} \Longrightarrow (q_{PA2}, t_{PA}, {q_{PA}}') \in R_{PA}) \end{array} \right) \end{aligned} \tag{4.10}$$

- For each transition $(q_{PA}, t_{PA}, {q_{PA}}') \in R_{PA}$, the transition relation R_{PA} contains all transitions to non-empty substates ${q_{PA}}''$ of ${q_{PA}}'$:

$$\begin{aligned} &\forall\, (q_{PA,} t_{PA}, {q_{PA}}') \in R_{PA}. \\ &\quad \forall\, {q_{PA}}'' \in Q_{PA}. \\ &\qquad {q_{PA}}'' \neq \emptyset \wedge {q_{PA}}'' \subseteq {q_{PA}}' \Longrightarrow (q_{PA}, t_{PA}, {q_{PA}}'') \in R_{PA} \end{aligned}$$

Closing a parcel automaton is a syntactic operation. It does not add new datapath behaviours.

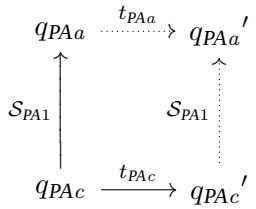
The following proposition states that we do not need Equation 4.9 in the definition of simulation for parcel automata if the parcel automata satisfy Equation 4.10.

4.6.3 Proposition. If ${pa}_c$ and ${pa}_a$ satisfy Equation 4.10 and there exists a simulation relation $\mathcal{S}_{PA}$ that satisfies Equation 4.7 and Equation 4.8 from the definition of simulation for parcel automata, then we can define $\mathcal{S}_{PA1}$ by extending $\mathcal{S}_{PA}$ such that $\mathcal{S}_{PA1}$ is a simulation for parcel automata.

Proof. We define $\mathcal{S}_{PA1}$ as follows:

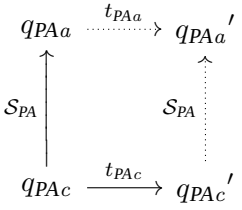
$$\begin{aligned} \mathcal{S}_{PA1} \equiv \mathcal{S}_{PA} \cup \{\, (q_{PAc}, q_{PAa}) \mid &\exists\, (q_{PAc\,1}, q_{PAa\,1}) \in \mathcal{S}_{PA}.\ \exists\, vars \subseteq \mathit{dom}\, q_{PAc\,1}. \\ &vars \neq \emptyset \wedge q_{PAc} = q_{PAc\,1} \mid_{vars} \wedge q_{PAa} = q_{PAa\,1} \mid_{vars} \} \end{aligned} \tag{4.11}$$

By construction, $\mathcal{S}_{PA}$ satisfies Equation 4.9. It also preserves labeling of states (Equation 4.7). We need to show it is a simulation relation that preserves transition labeling.



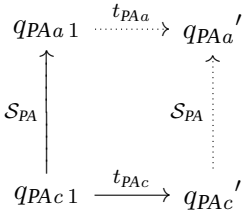
(4.12)

Case 1 $(q_{PAa}, q_{PAc}) \in \mathcal{S}_{PA}$. Equation 4.12 is equivalent to Equation 4.13 which holds because $\mathcal{S}_{PA}$ is a simulation relation.



(4.13)

Case 2 According to Equation 4.11 there exist $q_{PAc\,1} \in Q_{PAc}$, $q_{PAa\,1} \in Q_{PAa}$ and $vars \subseteq dom\, q_{PAc\,1}$ such that $q_{PAc} = q_{PAc\,1} \mid_{vars}$ and $q_{PAa} = q_{PAa\,1} \mid_{vars}$. Since pa_c is closed we have $(q_{PAc\,1}, t_{PAc}, q_{PAc}{}') \in R_{PAc}$. Since $(q_{PAc\,1}, q_{PAa\,1}) \in \mathcal{S}_{PA}$ the following diagram commutes:



(4.14)

Since

$$\begin{aligned} (q_{PAc}, t_{PAc}, q_{PAc}{}') &\in R_{Pc} \\ (q_{PAa\,1}, t_{PAa}, q_{PAa}{}') &\in R_{Pa} \\ q_{PAa} &\subseteq q_{PAa\,1} \end{aligned}$$

it follows that the transition $(q_{PAa}, t_{PAa}, q_{PAa}{}')$ is well defined and since pa_a is closed we have

$$(q_{PAa}, t_{PAa}, q_{PAa}{}') \in R_{PAa} \tag{4.15}$$

Combining Equation 4.15 with the fact that $\mathcal{S}_{PA} \subseteq \mathcal{S}_{PA1}$ we obtain Equation 4.12.

□

Simulation holds between the abstract parcel automaton and the concrete parcel automaton in Figure 4.28 and Figure 4.29 that models the concrete parcel $p_{\mathbf{A}}$. The simulation relation $\mathcal{S}_{PA}$ is defined by the following concrete-abstract pairs:

Concrete	Abstract
$\emptyset$	$\emptyset$
$r_{Neg} = \langle -40, 5, \texttt{mult} \rangle$	$r_{Neg} = \alpha_1$
$r_{Mult1} = \langle 40, 5 \rangle$	$r_{Mult1} = \alpha_4$
$r_{Mult2} = \langle 0, 0 \rangle$	$r_{Mult1} = \alpha_5$
$r_{Mult1} = \langle 40, 5 \rangle$	$r_{Mult1} = \alpha_6$
$r_{Mult2} = \langle a \times c = 8 \times 5 \rangle$	$r_{Mult2} = \alpha_7$
$r_{Mult1} = \langle 40, 5 \rangle$	$r_{Mult1} = \alpha_6$
$r_{Mult2} = \langle a \times c = 8 \times 5, \mathit{low}(bc) = 10 \rangle$	$r_{Mult2} = \alpha_7$
final_{PA}	final_{PA}

Language containment is defined with respect to the state and transition labels of the parcel automata. Parcel runs are denoted by $\sigma_{PA} \in \mathcal{L}(pa)$. For $k \in \mathbf{N}$ we use the notation:

$$\sigma_{PA}(k) = (q^k_{PA}, t^k_{PA})$$

Two runs $\sigma_{PAc} \in \mathcal{L}(pa_c)$ and $\sigma_{PAa} \in \mathcal{L}(pa_a)$ are equivalent if they specify states and transitions that have the same label. We overload the '$=_{PA}$' operator to denote equivalent runs:

$$\sigma_{PAc} =_{PA} \sigma_{PAa} \equiv \forall\, k \in \mathbf{N}.\ q^k_{PAc} =_{PA} q^k_{PAa} \wedge t^k_{PAc} =_{PA} t^k_{PAa}$$

Language containment is defined using run equivalence:

$$\begin{aligned} &\mathcal{L}(pa_c) \subseteq_{PA} \mathcal{L}(pa_a) \\ &\quad \equiv \\ &\qquad \forall\, \sigma_{PAc} \in \mathcal{L}(pa_c).\ \exists\, \sigma_{PAa} \in \mathcal{L}(pa_a).\ \sigma_{PAc} =_{PA} \sigma_{PAa} \end{aligned}$$

Recalling the example used for simulation between parcel automata, for the run of the parcel automaton that represents parcel $p_{\mathbf{A}}$ there exists an equivalent run of the abstract automaton shown in Figure 4.35.

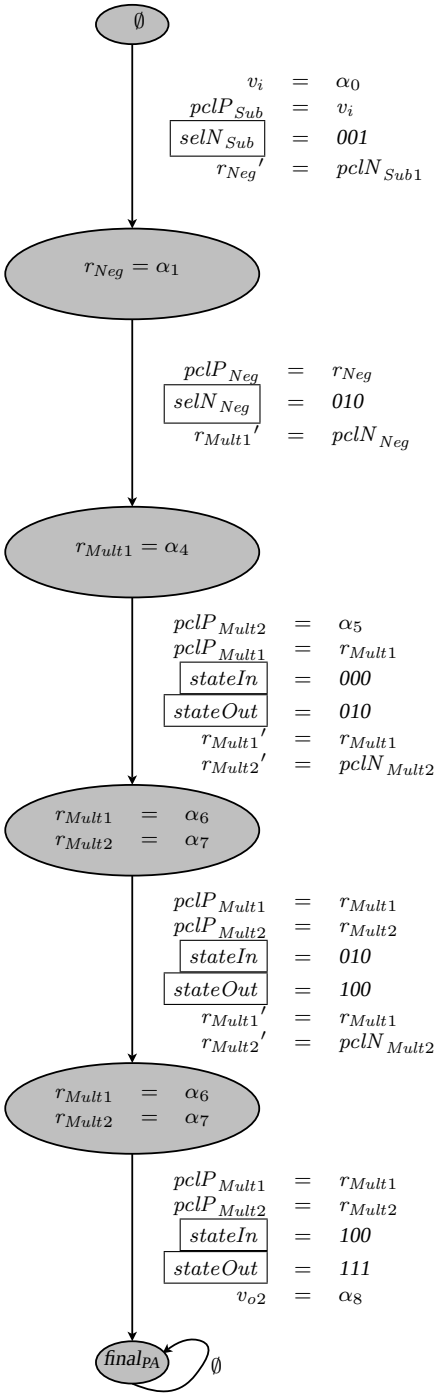


Figure 4.35: Abstract equivalent run corresponding to $p_{\mathbf{A}}$.

4.7 Abstract Interpretation Using Parcel Automata

In this section, we describe how we can obtain abstract interpretations of the concrete pipeline model using abstract parcel automata. In the previous sections, parcel automata were used to represent datapath behaviour of the pipeline model. In this section, we examine how the datapath behaviour represented by parcel automata can be used to define datapath circuits. As a direct application, abstract interpretations of the concrete pipeline model can be obtained by deriving abstract datapaths from abstract parcel automata.

We recall the equation that was used to define a consistent parcel step $(q_{PA}, \langle \mathit{fanOut}\ p, e_{pcl}, e_{ctrl} \rangle, {q_{PA}}')$:

$$\begin{array}{l} \forall\ dp \in \mathit{datapaths}\ (\mathit{fanOut}\ p). \\ \quad \exists\ (q_D, t_D, q_D') \in \mathit{LTS}(dp.C).R_C. \\ \quad\quad \forall\ v \in dp.V_{pcl} \cup dp.V_{ctrl}. \\ \quad\quad\quad (e_{ctrl} \cup e_{pcl})(\mathit{Arg}(v)) = t_D(v) \end{array} \tag{4.16}$$

Equation 4.16 characterizes parcel automaton transitions in terms of datapath behaviour. By reformulating it, we characterize datapath behaviour in terms of the parcel automaton. We use the parcel automaton to define the transition relation R_D of a labeled transition system that corresponds to a combinational datapath dp. The transition relation R_D is the union of all datapath behaviours that are specified in the parcel transitions of the parcel automaton:

$$\begin{array}{l} R_D \equiv \{ (\emptyset, t_D, \emptyset)\ \mid \\ \quad \exists\ (q_{PA}, \langle fg, e_{pcl}, e_{ctrl} \rangle, {q_{PA}}') \in R_{PA}. \\ \quad\quad dp \in \mathit{datapaths}\ (\mathit{fanOut}\ p) \wedge \forall\ v \in dp.V_{pcl} \cup dp.V_{ctrl}. \\ \quad\quad\quad t_D(v) = (e_{ctrl} \cup e_{pcl})(\mathit{Arg}(v))\ \} \end{array} \tag{4.17}$$

We illustrate Equation 4.17 using the abstract parcel automaton in Figure 4.33. There are three transitions of the abstract parcel automaton that mention arguments to the Sub datapath. We write

$$R_{Sub} \equiv \{\ (\emptyset, t_1, \emptyset), (\emptyset, t_2, \emptyset), (\emptyset, t_3, \emptyset)\ \}$$

And the three transition labels are defined by:

$$\left(\begin{array}{lcl} t_1(pclP_{Sub}) & = & \alpha_0 \\ t_1(selN_{Sub}) & = & \mathit{010} \\ t_1(pclN_{Sub1}) & = & \alpha_2 \\ t_1(pclN_{Sub2}) & = & \alpha_2 \end{array}\right) \wedge \left(\begin{array}{lcl} t_2(pclP_{Sub}) & = & \alpha_0 \\ t_2(selN_{Sub}) & = & \mathit{001} \\ t_2(pclN_{Sub1}) & = & \alpha_1 \\ t_2(pclN_{Sub2}) & = & \alpha_1 \end{array}\right) \wedge \left(\begin{array}{lcl} t_3(pclP_{Sub}) & = & \alpha_0 \\ t_3(selN_{Sub}) & = & \mathit{100} \\ t_3(pclN_{Sub1}) & = & \alpha_4 \\ t_3(pclN_{Sub2}) & = & \alpha_4 \end{array}\right)$$

```
1  type in_ty is { α0 };
2  type out_ty is { α1, α2, α4 };
3
4  ckt Sub_a(pclP : in_ty)(pclN_1 : out_ty, pclN_2 : out_ty, selN : bitvec[2])
5    var
6      case : 1 .. 3;
7    assign
8      case := choice;
9      pclN_1 :=
10       if case == 1 then α2
11       else if case == 2 then α1
12       else α4;
13     pclN_2 :=
14       if case == 1 then α2
15       else if case == 2 then α1
16       else α4;
17     selN :=
18       if case == 1 then 010
19       else if case == 2 then 001
20       else 100;
21 end
```

Figure 4.36: Circuit equivalent to R_{Sub}.

The transition relation R_{Sub} is represented by the circuit in Figure 4.36. We can similarly obtain the circuits Neg_a, Add_a and $Mult_a$ that are characterized by the transitions of the abstract parcel automaton. We have arrived at the point where we can use the abstract parcel automaton to give an abstract interpretation of the concrete pipeline datapath. We denote the datapaths created from the parcel automaton pa_a by *Dps* pa_a. The abstract interpretation obtained this way is denoted as:

$$Pipe_a = Pipe_c \left[\textit{Dps}\ pa_a / Dps_c \right]$$

The abstract parcel automaton defines the abstract datapaths. To perform an abstract interpretation we also replace occurrences of constants in the concrete expressions by abstract counterparts. If the abstract interpretation in Equation 4.7 holds, then the abstract parcel automaton is consistent with respect to constants.

4.8 Summary

Parcel automata formalize the behaviour of groups of related data values, called parcels, with respect to the pipeline datapath. Abstract parcel automata lead to the definition of abstract datapaths which are used to create abstract pipeline models using abstract interpretation. The conditions and correctness of datapath abstraction using parcel automata are described in the the next chapter.

Chapter 5

Datapath Abstraction Framework

Section 5.1 presents parcel maps and the concept of parcel independence that allows the runs of the pipeline model to be decomposed into runs of the parcel automaton. Section 5.2 describes the obligations needed to prove a parcel map satisfies the conditions of parcel independence. Section 5.3 presents additional requirements that must hold of the concrete pipeline model in order to apply abstraction using parcel automata. The soundness of abstraction using parcel automata is proven in Section 5.4.

5.1 Parcel Independence

In the previous chapter we described parcel automata as a formalism for the representation of parcel computations. One of the examples showed the simultaneous parcel computations that were taking place in the first few steps of a pipeline computation of the *DiffAddMult* model. Parcel independence is a property of pipeline computations that states that the computations can be decomposed into parcel computations that interact only through control variables. The parcel computations are independent of each other if they do not simultaneously use the same parcel variables or datapaths.

In the following we formalize the decomposition of pipeline runs into independent parcel computations. We first define the decomposition of a pipeline step into parcel steps and then state the condition under which the parcel steps form parcel computations. Our approach uses a function

$$PclMap : R_P \rightarrow \mathcal{P}(\mathcal{P}(V_{pcl} \cup \mathit{NextRegPcl}))$$

that returns the set of disjoint parcels at each step of the pipeline model. There are three properties that the parcel map must satisfy.

- Every parcel variable belongs to a parcel's fan-out.

$$\begin{array}{l} \forall\, (q_P, t_P, q_P') \in R_P. \\ \quad \forall\, v \in V_{pcl} \cup \mathit{NextRegPcl}.\ \exists\, p \in PclMap\ (q_P, t_P, q_P').\ v \in p^* \end{array} \tag{5.1}$$

- Datapaths transform only one parcel at a time.

$$\begin{array}{l} \forall\, (q_P, t_P, q_P') \in R_P. \\ \quad \forall\, dp \in Dps.\ \forall\, p_1 \in PclMap\ (q_P, t_P, q_P').\ \forall\, p_2 \in PclMap\ (q_P, t_P, q_P'). \\ \quad\quad \mathit{Arg}(dp.\mathit{PclP}) \cap p_1^* \neq \emptyset \wedge \mathit{Arg}(dp.\mathit{PclP}) \cap p_2^* \neq \emptyset \Longrightarrow p_1 = p_2 \end{array} \tag{5.2}$$

- The state of a parcel in the current step is part of the next state of a parcel in the previous step.

$$\begin{array}{l} \forall\, q_P.\ \forall\, t_P.\ \forall\, {q_P}'.\ \forall\, {t_P}'.\ \forall\, {q_P}''. \\ \quad (q_P, t_P, q_P') \in R_P \wedge ({q_P}', {t_P}', {q_P}'') \in R_P \Longrightarrow \\ \quad\quad \forall\, p_2 \in PclMap\ ({q_P}', {t_P}', {q_P}'').\ \exists\, p_1 \in PclMap\ (q_P, t_P, q_P'). \\ \quad\quad\quad \mathit{pclState}\ p_2 \subseteq \mathit{pclNextState}\ p_1 \end{array} \tag{5.3}$$

The first two properties of parcel maps ensure that the datapath computations in a pipeline step decompose into disjoint parcel steps. Since parcels are disjoint, the only way their fan-outs could not be disjoint is if two input arguments of a datapath were to be in the fan-out of distinct parcels. The inductive application of the third property ensures that parcel steps are connected into parcel computations.

parcel	condition
$\{\, v_i \,\}$	$\neg acc_{Mult,Sub}$
$\{\, v_i, {r_{Mult2}}' \,\}$	$acc_{Mult,Sub}$
$\{\, r_{Neg} \,\}$	$\neg acc_{Mult,Neg}$
$\{\, r_{Neg}, {r_{Mult2}}' \,\}$	$acc_{Mult,Neg}$
$\{\, {r_{Neg}}' \,\}$	$\neg(acc_{Neg,Sub} \vee stall_{Neg})$
$\{\, r_{Add} \,\}$	**true**
$\{\, {r_{Add}}' \,\}$	$\neg(acc_{Add,Sub} \vee acc_{Add,Neg} \vee stall_{Add})$
$\{\, r_{Mult1}, r_{Mult2} \,\}$	**true**
$\{\, {r_{Mult1}}', {r_{Mult2}}' \,\}$	$\neg(acc_{Mult,Sub} \vee acc_{Mult,Neg} \vee acc_{Mult,Mult})$

Figure 5.1: Parcel map for *DiffAddMult*.

For *DiffAddMult* the parcel map is defined in Figure 5.1. In the figure, a given parcel p belongs

to the parcel map if the pipeline step satisfies the corresponding condition. For instance, given the pipeline step (q_P^1, t_P^1, q_P^2) described in Figure 4.13, since

$$(q_P^1, t_P^1, q_P^2) \models (\neg acc_{Mult,Sub}) \wedge (acc_{Mult,Neg}) \wedge (\neg(acc_{Neg,Sub} \vee stall_{Neg}))$$

the parcel map returns the following value:

$$PclMap\ (q_P^1, t_P^1, q_P^2) = \{\ \{\ v_i\ \}, \{\ r_{Neg}, r_{Mult2}{}'\ \}, \{\ r_{Neg}{}'\ \}, \{\ r_{Add}\ \}, \{\ r_{Mult1}, r_{Mult2}\ \}\ \}$$

Examining the three properties satisfied by parcel maps we notice that the last two hold vacuously for singleton parcels. Thus they are automatically satisfied by parcel maps that only return singleton parcels. In addition, if every datapath has at most one input parcel variable, then the pipeline model is guaranteed to have a parcel map. We define a parcel map which returns singleton parcels and thus only needs to satisfy the first property. The singleton $\{\ v\ \}$ is a parcel if it is not in the fan-out of another parcel variable:

$$\{\ v\ \} \in PclMap\ (q_P, t_P, q_P') \iff \left(\begin{array}{c} v \in \mathit{RegPcl} \\ \vee \\ \exists\, (w, b, v) \in FanOutEdges.\ (q_P, t_P, q_P') \models b \\ \vee \\ \exists\, (\mathbf{choice}, b, v) \in FanOutEdges.\ (q_P, t_P, q_P') \models b \\ \vee \\ \exists\ dp \in Dps.\ |dp.\mathit{PclP}| = 0 \wedge v \in \mathit{Arg}(dp.\mathit{PclN}) \end{array} \right) \tag{5.4}$$

The existence of a parcel map for a pipeline model allows us to perform datapath abstraction using parcel automata. A parcel map induces a parcel automaton which we use to reason about datapath abstractions. The transitions of the automaton correspond to the parcel steps induced by the parcel map.

5.1.1 Definition (Parcel Automaton Induced by Parcel Map)**.** A parcel map $PclMap$ induces a parcel automaton $\mathit{pa}(Pipe, PclMap) = \langle Q_{PA}, R_{PA}, T_{PA}, I_{PA} \rangle$ as follows:

$$Q_{PA} = \{\ q_{PA} \mid q_{PA} \in \mathit{PEnv}(\mathit{RegPcl})\ \} \tag{5.5}$$

$$\begin{aligned} R_{PA} = {} & \{\ (\mathit{final}_{PA}, \emptyset, \mathit{final}_{PA})\ \} \cup \\ & \{\ (q_{PA}, t_{PA}, q_{PA}{}') \mid \\ & \quad \exists\, (q_P, t_P, q_P') \in R_P.\ \exists\ p \in PclMap\ (q_P, t_P, q_P'). \\ & \quad\quad q_{PA} = q_P \mid_p \wedge\ t_{PA} = \mathit{pclTrans}\ p \wedge q_{PA}{}' \subseteq \mathit{pclNextState}\ p\ \} \end{aligned} \tag{5.6}$$

$$I_{PA} = \{\ \emptyset\ \} \cup \{\ q_{PA} \mid \exists\, (q_P, t_P, q_P') \in R_P.\ q_P \in I_P \wedge$$

$$\exists\, p \in PclMap\,(q_P, t_P, q'_P).\ q_{PA} = \mathit{pclState}\ p\,\}\tag{5.7}$$

5.2 Verification Of Parcel Independence

In this section we describe how we can state the three properties of well-defined maps into propositional logic. For each property, we describe a propositional formula that is a tautology if and only if the property is true.

We use an equivalent representation of the parcel map

$$PclMap_{alt} : R_P \to V_{pcl} \cup \mathit{NextRegPcl} \to \{\, 0, \ldots, |V_{pcl} \cup \mathit{NextRegPcl}| \,\}$$

so that a parcel variable maps to a non-zero value if it belongs to a parcel

$$\begin{array}{l}\forall\,(q_P, t_P, q'_P) \in R_P.\\ \quad \forall\, v \in V_{pcl} \cup \mathit{NextRegPcl}.\\ \quad\quad PclMap_{alt}\,(q_P, t_P, q'_P)\ v \neq 0 \iff \exists\, p \in PclMap\,(q_P, t_P, q'_P).\ v \in p\end{array}$$

and two parcel variables map to the same non-zero value if they belong to the same parcel.

$$\begin{array}{l}\forall\,(q_P, t_P, q'_P) \in R_P.\\ \quad \forall\, v_1 \in V_{pcl} \cup \mathit{NextRegPcl}.\ \forall\, v_2 \in V_{pcl} \cup \mathit{NextRegPcl}.\\ \quad\quad PclMap_{alt}\,(q_P, t_P, q'_P)\ v_1 \neq 0 \wedge PclMap_{alt}\,(q_P, t_P, q'_P)\ v_1 = PclMap_{alt}\ v_2\\ \quad\quad\quad \iff\\ \quad\quad\quad\quad \exists\, p \in PclMap\,(q_P, t_P, q'_P).\ \{\, v_1, v_2 \,\} \subseteq p\end{array}$$

Recall from Section 2.3 that the transition relation R_C of a circuit C is represented by a propositional formula $\mathit{formula}(\mathit{Elab}(C).\mathit{Tr})$. For a pipeline model its transition relation R_P is represented by the formula $\mathit{formula}(\mathit{Elab}(Pipe.C).\mathit{Tr})$ which we designate by $[R_P]_{bool}$. This formula is defined over current-state register variables, combinational variables and next-state register variables, denoted in order as V_{reg}, V_{comb}, $V_{nextReg}$. To make explicit the variables that appear in the formula we write it as $[R_P]_{bool}\,(V_{reg}, V_{comb}, V_{nextReg})$. Note that the set of variables V_{comb} subsumes the set of input parcel variables *InputPcl*. We also let V_{step} stand for $V_{reg} \cup V_{comb} \cup V_{nextReg}$.

The propositional formula that represents the parcel map is given by $[PclMap_{alt}]_{bool}\,(V_{step}, V_{pclMap})$, where V_{pclMap} is in bijection with $V_{pcl} \cup \mathit{NextRegPcl}$ and each variable $pclMap_v \in V_{pclMap}$ represents the value $PclMap_{alt}\,(q_P, t_P, q'_P)\ v$. The semantics of the propositional representation is

summarized by the equation below:

$$\begin{aligned}
&\forall\, q_P \in Q_P.\ \forall\, t_P \in T_P.\ \forall\, {q_P}' \in Q_P.\ \forall\, e_{pclMap} \in \mathit{Env}(V_{pclMap}).\\
&\quad (q_P \cup t_P \cup {q_P}' \left[V_{nextReg} / V_{reg} \right] \cup e_{pclMap}) \models [\![R_P]\!]_{bool}\,(V_{step}) \wedge [\![PclMap_{alt}]\!]_{bool}\,(V_{step},\, V_{pclMap})\\
&\qquad \Longleftrightarrow\\
&\qquad\quad (q_P, t_P, {q_P}') \in R_P \wedge \forall\, v \in V_{pcl} \cup \mathit{NextRegPcl}.\ PclMap_{alt}\ (q_P, t_P, q'_P)\ v = e_{pclMap}(pclMap_v)
\end{aligned}$$

Next, we describe how we represent the fan-out of parcels in propositional logic. We use the set of fan-out variables V_{fanOut} in bijection with the set $V_{pcl} \cup \mathit{NextRegPcl}$ to represent for each parcel variable the parcel that contains it in its fan-out. Thus, $fanOut_v = 0$ means that the parcel variable v is not in the fan-out of any parcel, while $fanOut_v = n$, with $n \neq 0$ means that $v \in p^*$ with $p = \{\, v_1 \mid PclMap_{alt}\ (q_P, t_P, q'_P)\ v_1 = n \,\}$.

A parcel's fan-out is derived transitively using fan-out edges. A fan-out edge $(v_l, b, v_k) \in FanOutEdges$ corresponds to the assignment $fanOut_{v_k} := fanOut_{v_l}$. Since the assignment is performed only when b holds, the propositional formula for the fan-out edge becomes

$$b \Longrightarrow (fanOut_{v_k} := fanOut_{v_l})$$

Similarly, edges of form (w, b, v_k), with w constant, and $(\mathbf{choice}, b, v_k)$ are encoded as

$$b \Longrightarrow (fanOut_{v_k} := pclMap_{v_k})$$

Since every combinational and next-state register parcel variable is assigned, for each such variable there must exist an incoming fan-out edge that is satisfied under any control environment.

The propositional formula that describes the fan-out of parcels is defined as follows:

$$FanOut(V_{step}, V_{pclMap}, V_{fanOut}) \equiv \left(\begin{array}{c}
\bigwedge\limits_{v \in \mathit{InputPcl}} (fanOut_v := pclMap_v)\\
\wedge\\
\bigwedge\limits_{v \in V_{reg}} (fanOut_v := pclMap_v)\\
\wedge\\
\bigwedge\limits_{(v_l, b, v_k) \in FanOutEdges} b \Longrightarrow (fanOut_{v_k} := fanOut_{v_l})\\
\wedge\\
\bigwedge\limits_{(w, b, v_k) \in FanOutEdges} b \Longrightarrow (fanOut_{v_k} := pclMap_{v_k})\\
\wedge\\
\bigwedge\limits_{(\mathbf{choice}, b, v_k) \in FanOutEdges} b \Longrightarrow (fanOut_{v_k} := pclMap_{v_k})
\end{array} \right)$$

We formulate the first two properties of a well-defined parcel map as formulas of propositional logic.

- Every parcel variable belongs to a parcel's fan-out.

$$\begin{aligned}
&[\![R_P]\!]_{bool}\,(V_{step}) \wedge [\![PclMap_{alt}]\!]_{bool}\,(V_{step}, V_{pclMap}) \wedge FanOut(V_{step}, V_{pclMap}, V_{fanOut}) \\
&\quad \Longrightarrow \\
&\qquad \bigwedge_{v \in V_{pcl} \cup \mathit{NextRegPcl}} fanOut_v \neq 0
\end{aligned} \tag{5.8}$$

- Datapaths transform only one parcel at a time.

$$\begin{aligned}
&[\![R_P]\!]_{bool}\,(V_{step}) \wedge [\![PclMap_{alt}]\!]_{bool}\,(V_{step}, V_{pclMap}) \wedge FanOut(V_{step}, V_{pclMap}, V_{fanOut}) \\
&\quad \Longrightarrow \\
&\qquad \bigwedge_{dp \in Dps} \left(\bigwedge_{\{\, pclP_1, pclP_2 \,\} \subseteq dp.\mathit{PclP}} fanOut_{pclP_1} = fanOut_{pclP_2} \right)
\end{aligned} \tag{5.9}$$

The third property, stated in Equation 5.3 can be reformulated as follows:

$$\begin{aligned}
&\forall\, q_P.\ \forall\, t_P.\ \forall\, {q_P}'.\ \forall\, {t_P}'.\ \forall\, {q_P}''. \\
&\quad (q_P, t_P, q'_P) \in R_P \wedge (q'_P, t'_P, q''_P) \in R_P \\
&\qquad \Longrightarrow \\
&\qquad \left(\begin{array}{l}
\forall\, v_1 \in \mathit{RegPcl}.\ \forall\, v_2 \in \mathit{RegPcl}. \\
\quad \exists\, p_2 \in PclMap\,(q'_P, t'_P, q''_P).\ \{\, v_1, v_2 \,\} \subseteq p_2 \\
\qquad \Longrightarrow \\
\qquad\quad \exists\, p_1 \in PclMap\,(q_P, t_P, q'_P).\ \{\, {v_1}', {v_2}' \,\} \subseteq p_1^*
\end{array} \right)
\end{aligned} \tag{5.10}$$

In order to state Equation 5.10 into an equivalent propositional logic formula, we need two copies of each of the following sets of variables: V_{step}, V_{pclMap} and V_{fanOut}. We will denote the two copies of V by V^1 and V^2. For $v \in V$ we denote its two copies as $v^1 \in V^1$ and $v^2 \in V^2$. The propositional formula for Equation 5.10 is defined below.

- The state of a parcel in the current step is part of the state of a parcel in the previous step.

$$\left(\begin{array}{c} [\![R_P]\!]_{bool}\,(V_{step}{}^1) \wedge [\![PclMap_{alt}]\!]_{bool}\,(V_{step}{}^1, V_{pclMap}{}^1) \wedge FanOut(V_{step}{}^1, V_{pclMap}{}^1, V_{fanOut}{}^1) \\ \wedge \\ [\![R_P]\!]_{bool}\,(V_{step}{}^2) \wedge [\![PclMap_{alt}]\!]_{bool}\,(V_{step}{}^2, V_{pclMap}{}^2) \wedge FanOut(V_{step}{}^2, V_{pclMap}{}^2, V_{fanOut}{}^2) \\ \wedge \\ \bigwedge_{v \in V_{reg}} ((v^1)' = v^2) \end{array}\right)$$

$$\Longrightarrow$$

$$\bigwedge_{v_1^2 \in RegPcl^2} \bigwedge_{v_2^2 \in RegPcl^2} (pclMap^2_{v_1^2} = pclMap^2_{v_2^2} \Longrightarrow fanOut^1_{(v_1^1)'} = fanOut^1_{(v_2^1)'}) \tag{5.11}$$

5.3 Concrete Pipeline Models And Abstract Interpretation

In Section 3.3 we describe the general form of abstract interpretation of pipeline models and in Section 4.7 we explain how abstract parcel automata are used to derive abstract datapaths suitable for abstract interpretation. In this section we examine the relationship between concrete and abstract pipeline model states and transitions.

5.3.1 Assumptions About Initial States

We write the initial conditions of the concrete and abstract models as disjoint unions between the set of assignments to control variables and the set of assignments to parcel variables.

$$\begin{aligned} Pipe_c.Init &= Init_{ctrl} \uplus Init_{pcl\,c} \\ Pipe_a.Init &= Init_{ctrl} \uplus Init_{pcl\,a} \end{aligned}$$

In the abstract pipeline model any state that is a disjoint union of initial states of the parcel automaton satisfies the initial parcel variable constraint.

$$\forall\, q_{Pa} \in Q_{Pa}.$$
$$\left(\exists\, P \in \mathcal{P}(RegPcl).\ RegPcl = \biguplus_{p \in P} p \wedge \forall\, p \in P.\ q_{Pa} \mid_p \in I_{PAa}\right) \Longrightarrow q_{Pa} \models Init_{pcl\,a} \tag{5.12}$$

We require all abstract interpretations to satisfy Equation 5.12. We note that Equation 5.12 holds trivially if $Init_{pcl\,a}$ is empty.

The concept of induced parcel states is used to describe a property of the parcel map with respect to initial concrete model states. It will also appear when we construct the simulation between the concrete and abstract pipeline models. Given a concrete pipeline model step $(q_{Pc}, t_{Pc}, q'_{Pc}) \in R_{Pc}$ the parcel map induces a set of parcel automaton states $\mathit{PclStates}\,(q_{Pc}, t_{Pc}, q'_{Pc}) \subseteq Q_{PAc}$. We use $\mathit{PclStates}\,q_{Pc}$ to denote the set of states defined as the union of $\mathit{PclStates}\,(q_{Pc}, t_{Pc}, q'_{Pc})$ over all steps $(q_{Pc}, t_{Pc}, q'_{Pc})$ from q_{Pc}.

$$\begin{aligned} \mathit{PclStates}\,(q_{Pc}, t_{Pc}, q'_{Pc}) &\equiv \{\, \mathit{pclState}\; p \mid p \in \mathit{PclMap}\,(q_{Pc}, t_{Pc}, {q_{Pc}}') \wedge p \cap \mathit{RegPcl} \neq \emptyset \,\} \\ \mathit{PclStates}\,q_{Pc} &\equiv \bigcup_{(q_{Pc}, t_{Pc}, q'_{Pc}) \in R_{Pc}} \mathit{PclStates}\,(q_{Pc}, t_{Pc}, q'_{Pc}) \end{aligned}$$

We say that the parcel states are fixed in a state $q_{Pc} \in Q_{Pc}$ if in all steps from state q_{Pc} the parcel map induces the same set of states:

$$\begin{array}{l} \forall\, t_{Pc\,1}.\ \forall\, q_{Pc\,1}.\ \forall\, t_{Pc\,2}.\ \forall\, q_{Pc\,2}. \\ \quad (q_{Pc}, t_{Pc\,1}, q_{Pc\,1}) \in R_{Pc} \wedge (q_{Pc}, t_{Pc\,2}, q_{Pc\,2}) \in R_{Pc} \\ \qquad \Longrightarrow \\ \qquad\quad \mathit{PclStates}\,(q_{Pc}, t_{Pc\,1}, q_{Pc\,1}) = \mathit{PclStates}\,(q_{Pc}, t_{Pc\,2}, q_{Pc\,2}) \end{array} \tag{5.13}$$

Our proof of simulation between the abstract and concrete pipeline models relies on Equation 5.13 to hold for initial states. We therefore provide a way to verify it by giving a translation into propositional logic. Equation 5.13 rewrites alternatively as follows:

$$\begin{array}{l} \forall\, (q_{Pc\,1}, t_{Pc\,1}, q'_{Pc\,1}) \in R_{Pc}.\ \forall\, (q_{Pc\,2}, t_{Pc\,2}, q'_{Pc\,2}) \in R_{Pc}. \\ \quad q_{Pc\,1} = q_{Pc\,2} \wedge q_{Pc\,1} \in I_{Pc} \\ \qquad \Longrightarrow \\ \qquad \left(\begin{array}{l} \forall\, p_1 \in \mathit{PclMap}\,(q_{Pc\,1}, t_{Pc\,1}, q'_{Pc\,1}). \\ \quad p_1 \cap \mathit{RegPcl} \neq \emptyset \\ \qquad \Longrightarrow \\ \qquad\quad \exists\, p_2 \in \mathit{PclMap}\,(q_{Pc\,2}, t_{Pc\,2}, q'_{Pc\,2}). \\ \qquad\qquad p_1 \cap \mathit{RegPcl} = p_2 \cap \mathit{RegPcl} \end{array} \right) \end{array} \tag{5.14}$$

Equation 5.14 states that, for any two pipeline model steps from an initial state q_{Pc}, the set of parcel states of the former is a subset of the set of parcel states of the latter. This is equivalent to stating that the sets of parcel states of any two pipeline model steps are the same and equal to $\mathit{PclStates}\,q_{Pc}$. We show how Equation 5.14 expresses in term of the alternate representation of the parcel map

$PclMap_{alt}$:

$$\forall\,(q_{Pc\,1}, t_{Pc\,1}, q'_{Pc\,1}) \in R_{Pc}.\ \forall\,(q_{Pc\,2}, t_{Pc\,2}, q'_{Pc\,2}) \in R_{Pc}.$$
$$q_{Pc\,1} = q_{Pc\,2} \wedge q_{Pc\,1} \in I_{Pc}$$
$$\Longrightarrow$$
$$\bigwedge_{v_1 \in RegPcl} \bigwedge_{v_2 \in RegPcl} \left(\begin{array}{l} PclMap_{alt}\,(q_{Pc\,1}, t_{Pc\,1}, q'_{Pc\,1})\ v_1 = PclMap_{alt}\,(q_{Pc\,1}, t_{Pc\,1}, q'_{Pc\,1})\ v_2 \\ \quad \Longrightarrow \\ \quad PclMap_{alt}\,(q_{Pc\,2}, t_{Pc\,2}, q'_{Pc\,2})\ v_1 = PclMap_{alt}\,(q_{Pc\,2}, t_{Pc\,2}, q'_{Pc\,2})\ v_2 \end{array} \right) \tag{5.15}$$

Using the approach described in Section 5.2, Equation 5.15 is represented in propositional logic by the following formula:

$$\left(\begin{array}{c} [\![R_P]\!]_{bool}\,(V_{step}{}^1) \wedge [\![PclMap_{alt}]\!]_{bool}\,(V_{step}{}^1, V_{pclMap}{}^1) \\ \wedge \\ [\![R_P]\!]_{bool}\,(V_{step}{}^2) \wedge [\![PclMap_{alt}]\!]_{bool}\,(V_{step}{}^2, V_{pclMap}{}^2) \\ \wedge \\ \bigwedge_{v \in V_{reg}} (v^{1\prime} = v^2) \\ \wedge \\ [\![I_P]\!]_{bool}\,(V_{reg}{}^1) \end{array} \right) \tag{5.16}$$
$$\Longrightarrow$$
$$\bigwedge_{v_1^1 \in RegPcl^1} \bigwedge_{v_2^1 \in RegPcl^1} (pclMap^1_{v_1^1} = pclMap^1_{v_2^1}) \Longrightarrow (pclMap^2_{v_1^2} = pclMap^2_{v_2^2})$$

5.3.2 Fundamental Relationship

Theorem 5.3.1 states the mechanism by which a step of the concrete pipeline model is matched by a step of the abstract model. We recall the $AndOr$ example from Section 4.1. Figure 5.2 shows a pipeline model commuting diagram that is derived on the basis of parcel automata commuting diagrams. The parcel commuting diagrams describe abstract parcel steps that match the concrete parcel steps that happen within one concrete pipeline model step. Given a concrete pipeline model state and a control equivalent abstract pipeline model state such that the parcel diagrams commute, we can construct a matching abstract pipeline model step. The abstract pipeline model step is constructed using the abstract pipeline automaton steps.

5.3.1 Theorem. Let $Pipe_c$ and $Pipe_a$ be two pipeline models such that

$$Pipe_a = Pipe_c \left[Dps\ pa_a / Dps_c \right] \tag{5.17}$$

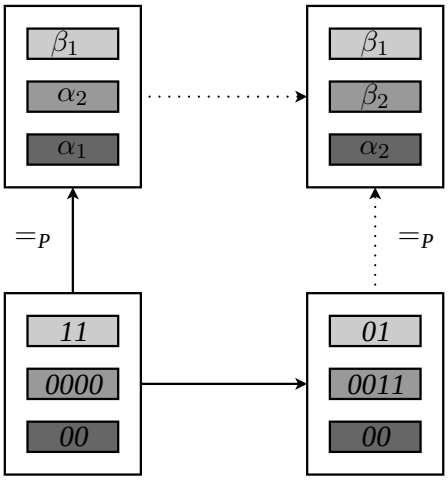


Fig. 5.2a. Pipeline commuting diagram.

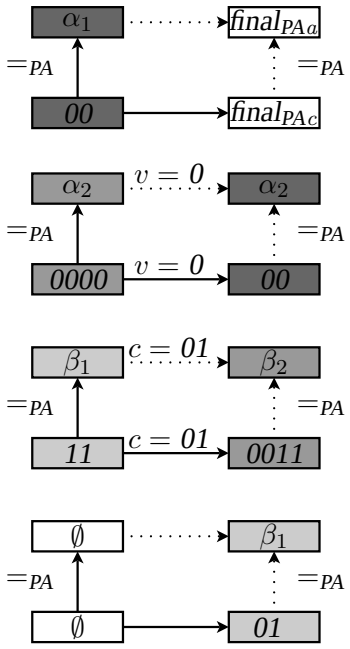


Fig. 5.2b. Parcel commuting diagrams.

Figure 5.2: $AndOr$ example.

Given

- $(q_{Pc}, t_{Pc}, q'_{Pc}) \in R_{Pc}$
- $q_{Pa} \in Q_{Pa}$
- $pa_a \in Pa(Pipe_a)$
- $pclTrans_a : PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc}) \rightarrow T_{PAa}$
- $pclNextState_a : PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc}) \rightarrow Q_{PAa}$

such that:

- The pipeline states q_{Pc} and q_{Pa} are control equivalent.

$$q_{Pc} =_{V_{ctrl}} q_{Pa} \tag{5.18}$$

- The following diagram commutes:

$$\begin{array}{l}\forall\, p \in PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc}).\\ \quad \begin{array}{ccc} q_{Pa} \mid p & \overset{pclTrans_a\ p}{\cdots\cdots\rightarrow} & pclNextState_a p \\ \Big\uparrow{\scriptstyle =_{PA}} & & \Big\uparrow{\scriptstyle =_{PA}} \\ q_{Pc} \mid p & \xrightarrow{pclTrans\ p} & pclNextState\ p \end{array}\end{array} \tag{5.19}$$

Then there exist $t_{Pa} \in T_{Pa}$ and $q_{Pa}{}' \in Q_{Pa}$ such that the following diagram commutes:

$$\begin{array}{ccc} q_{Pa} & \overset{t_{Pa}}{\cdots\cdots\rightarrow} & q_{Pa}{}' \\ \Big\uparrow{\scriptstyle =_{P}} & & \Big\uparrow{\scriptstyle =_{P}} \\ q_{Pc} & \xrightarrow{t_{Pc}} & q_{Pc}{}' \end{array} \tag{5.20}$$

Furthermore, t_{Pa} and $q_{Pa}{}'$ are constructed using $pclTrans_a$, and $pclNextState_a$. Let $pclTrans_a\ p = (fg_{a\,p}, e_{pcla\,p}, e_{ctrla\,p})$. The construction has the following properties:

$$\begin{array}{l}\forall\, p \in PclMap\ (q_P, t_P, q'_P).\\ \quad \forall\, v \in p^* \cap \textit{CombPcl}.\ t_{Pa}(v) = e_{pcla\,p}(v)\end{array} \tag{5.21}$$

$$\begin{array}{l}\forall\, p \in PclMap\ (q_P, t_P, q'_P).\\ \quad \forall\, v' \in p^* \cap \textit{NextRegPcl}.\ q_{Pa}{}'(v) = (pclNextState_a\ p)(v)\end{array} \tag{5.22}$$

Proof. Equation 5.21 defines t_{Pa} over parcel variables. Similarly, Equation 5.22 defines $q_{Pa}{}'$ over parcel variables. Since the diagram in Equation 5.20 commutes we also have:

$$t_{Pa} \quad =_{V_{ctrl}} \quad t_{Pc} \tag{5.23}$$

$$q_{Pa}{}' \quad =_{V_{ctrl}} \quad q_{Pc}{}' \tag{5.24}$$

Equation 5.21 and Equation 5.23 define $t_{Pa} \in T_{Pa}$ over the combinational pipeline variables. Equation 5.22 and Equation 5.24 define $q_{Pa}{}'$ over the entirety of its domain. We must show that t_{Pa} can be defined over the combinational instance variables so that $(q_{Pa}, t_{Pa}, q_{Pa}{}') \in R_{Pa}$. We apply Proposition 2.3.16. Accordingly, we have two obligations:

$$\begin{array}{l}\forall\, \text{`}v := expr_a\text{'} \in Pipe_a.C.Tr.\\ \quad (t_{Pa} \cup q_{Pa}{}' \left[V_{nextReg} / V_{reg} \right])(v) = [\![expr_a]\!]_{q_{Pa} \cup t_{Pa}}\end{array} \tag{5.25}$$

$$\begin{array}{l}\forall\ dp_a \in Pipe_a.\mathit{Dps}.\\ \quad \exists\ (q_{Da}, t_{Da}, {q_{Da}}') \in \mathit{LTS}(dp_a.C).R_C.\\ \quad\quad \forall\ v \in dp_a.C.V_i \cup dp_a.C.V_o.\ t_{Da}(v) = t_{Pa}(\mathit{Arg}(v))\end{array} \tag{5.26}$$

Part 1 Consider '$v := expr_a$' $\in Pipe_a.C.Tr$. There are two cases depending on whether v is a control variable or a parcel variable.

Part 1a $v \in (V_{ctrl} \cap V_c) \cup (V_{ctrl} \cap V_r)'$. Since $Pipe_a$ is an abstract interpretation and v is a control variable we have

$$\text{`}v := expr_a\text{'} \in Pipe_c.C.Tr \tag{5.27}$$

In this case $expr_a$ is an expression over control variables. Equation 5.18, Equation 5.23 and Equation 5.24 imply:

$$(t_{Pa} \cup {q_{Pa}}' \left[V_{nextReg} / V_{reg} \right])(v) = (t_{Pc} \cup {q_{Pc}}' \left[V_{nextReg} / V_{reg} \right])(v) \tag{5.28}$$

$$[\![expr_a]\!]_{q_{Pa} \cup t_{Pa}} = [\![expr_a]\!]_{q_{Pc} \cup t_{Pc}} \tag{5.29}$$

From Equation 5.27 it follows that

$$(t_{Pc} \cup {q_{Pc}}' \left[V_{nextReg} / V_{reg} \right])(v) = [\![expr_a]\!]_{q_{Pc} \cup t_{Pc}} \tag{5.30}$$

Equation 5.25 follows from Equation 5.28, Equation 5.29 and Equation 5.30.

Part 1b $v \in \mathit{CombPcl} \cup \mathit{NextRegPcl}$. According to Section 4.7, $expr_a$ is an ITE parcel expression that is either equal to a constant or contains only **choice** and parcel variables. Consider $p \in PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc})$ such that $v \in p^*$. We apply the commuting diagram in Equation 5.19 to parcel p. Letting

$$\begin{array}{rcl}\mathit{pclTrans}\ p & = & (fg_{c\,p}, e_{pclc\,p}, e_{ctrlc\,p})\\ pclTrans_a\ p & = & (fg_{a\,p}, e_{pcla\,p}, e_{ctrla\,p})\end{array}$$

we have

$$\begin{array}{rcl}fg_{c\,p} & = & fg_{a\,p}\\ e_{ctrlc\,p} & = & e_{ctrla\,p}\end{array} \tag{5.31}$$

Case 1 If $expr_a$ reduces to a constant w_a then the assignment $v := w_a$ corresponds to an abstract fan-out edge (w_a, b, v) on the transition $pclTrans_a\ p$ such that $v \in p$. Therefore

$$(e_{pcla\,p} \cup (pclNextState_a p) \left[\mathit{NextRegPcl} / \mathit{RegPcl} \right])(v) = w_a$$

and thus

$$[\![v]\!]_{e_{pcla\,p} \cup (pclNextState_a\,p)\left[\mathit{NextRegPcl}/\mathit{RegPcl}\right])} = [\![expr_a]\!]_{e_{pcla\,p} \cup (pclNextState_a\,p)\left[\mathit{NextRegPcl}/\mathit{RegPcl}\right])}$$

which implies Equation 5.25.

Case 2 If $expr_a$ reduces to **choice**

$$[\![expr_a]\!]_{(q_{Pa} \cup t_{Pa})|V_{ctrl}} = \mathbf{choice}$$

then any environment satisfies the assignment '$v := \mathbf{choice}$'.

Case 3 The final case is when $expr_a$ reduces to $u \in V_{pcl}$:

$$[\![expr_a]\!]_{(q_{Pa} \cup t_{Pa})|V_{ctrl}} = u$$

Consider the assignment '$v := expr_c$' $\in Pipe_c.C.Tr.$ Since $(q_{Pc} \cup t_{Pc}) \mid_{V_{ctrl}} = (q_{Pa} \cup t_{Pa}) \mid_{V_{ctrl}}$ and $expr_c \approx_{ai} expr_a$ then $expr_c$ must also reduce to u.

$$[\![expr_c]\!]_{(q_{Pc} \cup t_{Pc})|V_{ctrl}} = u$$

Therefore, $fg_{c\,p}$ contains an edge of form (u, b, v) such that $(q_{Pc} \cup t_{Pc}) \mid_{V_{ctrl}} \models b$. Equation 5.31 implies that the same edge exists in $fg_{a\,p}$. According to Equation 4.3 in the definition of a parcel step we must have

$$e_{pcla\,p}(v) = e_{pcla\,p}(u) \tag{5.32}$$

Equation 5.21 and Equation 5.22 imply that

$$(t_{Pa} \cup {q_{Pa}}'\left[V_{nextReg}/V_{reg}\right])(v) = e_{pcla\,p}(v) \tag{5.33}$$

Since $pclTrans_a\ p$ labels the transition from the parcel state $q_{Pa} \mid_p$ we also have:

$$(q_{Pa} \cup t_{Pa})(u) = e_{pcla\,p}(u) \tag{5.34}$$

Equation 5.32, Equation 5.33 and Equation 5.34 imply that

$$(t_{Pa} \cup {q_{Pa}}'\left[V_{nextReg}/V_{reg}\right])(v) = (q_{Pa} \cup t_{Pa})(u)$$

Part 2 Consider $dp_a \in Pipe_a.\mathit{Dps}$. Let $dp_c \in Pipe_c.\mathit{Dps}$ be the corresponding datapath given by the rules of abstract interpretation. There must exist a parcel $p \in PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc})$ such that

the input and output arguments of dp_c belong to p^*. The parcel's step expresses as:

$$\mathit{pclStep}\ p = \Big(\ q_{Pc} \mid_{p}, \mathit{pclTrans}\ p, pclNextState_a\ p\ \Big) \tag{5.35}$$

Applying the commuting diagram in Equation 5.19 to parcel p we have the matching step:

$$\Big(\ q_{Pa} \mid_{p}, pclTrans_a\ p, pclNextState_a p\ \Big) \tag{5.36}$$

Letting

$$\begin{aligned} \mathit{pclTrans}\ p &= (fg_{c\,p}, e_{pclc\,p}, e_{ctrlc\,p}) \\ pclTrans_a\ p &= (fg_{a\,p}, e_{pcla\,p}, e_{ctrla\,p}) \end{aligned}$$

we have

$$fg_{c\,p} = fg_{a\,p} \tag{5.37}$$

$$e_{ctrlc\,p} = e_{ctrla\,p} \tag{5.38}$$

The parcel step in Equation 5.35, satisfies Equation 4.1 and Equation 4.2 in the definition of the parcel step. Therefore, its fanout graph $fg_{c\,p}$ contains the parcel arguments of dp_c and the domain of $e_{ctrlc\,p}$ contains the control arguments of dp_c. Equation 5.37 and Equation 5.38 imply that $fg_{a\,p}$ contains the parcel arguments of dp_a and that the domain of $e_{ctrla\,p}$ contains the control arguments of dp_a. Therefore, the step in Equation 5.36 contains a computation of dp_a. Since pa_a is consistent with respect to $\mathit{Dps}\ pa_a$, applying Equation 4.4 we get:

$$\begin{aligned} &\exists\,(q_{Da}, t_{Da}, q'_{Da}) \in \mathit{LTS}(dp_a.C).R_C. \\ &\quad \forall\ v \in dp_a.V_{pcl} \cup dp_a.V_{ctrl}. \\ &\qquad (e_{ctrla\,p} \cup e_{pcla\,p})(\mathit{Arg}(v)) = t_{Da}(v) \end{aligned} \tag{5.39}$$

Since the parcel step in Equation 5.35 is part of $(q_{Pc}, t_{Pc}, q'_{Pc})$ we have

$$e_{ctrlc\,p} \subseteq t_{Pc} \mid_{V_{ctrl}}$$

and also

$$\begin{aligned} e_{ctrla\,p} &= e_{ctrlc\,p} \\ t_{Pc} \mid_{V_{ctrl}} &= t_{Pa} \mid_{V_{ctrl}} \end{aligned}$$

Combining the last two steps we get

$$e_{ctrl\,a\,p} \cup e_{pcl\,a\,p} \subseteq t_{PAa}$$

which is used to rewrite Equation 5.39 to

$$\begin{gathered}\exists\,(q_{Da}, t_{Da}, q'_{Da}) \in \mathit{LTS}(dp_a.C).R_C.\\ \forall\, v \in dp_a.V_{pcl} \cup dp_a.V_{ctrl}.\\ t_{Pa}(\mathsf{Arg}(v)) = t_{Da}(v)\end{gathered}$$

which is the desired conclusion.

□

5.4 General Correctness

We present two correctness theorems that link abstraction of parcel automata to abstraction of pipeline models. Theorem 5.4.1 states that simulation of parcel automata transfers to simulation of pipeline models and Theorem 5.4.8 states the similar result for language containment.

5.4.1 Simulation

5.4.1 Theorem (Abstraction Using Simulation)**.** Let $Pipe_c$ and $Pipe_a$ be two pipeline models such that

$$Pipe_c = Pipe_a\left[Dps_a/Dps_c\right] \tag{5.40}$$

If

- The abstract automaton simulates the induced parcel automaton:

$$pa(Pipe_c, PclMap) \preceq_{PA} pa_a \tag{5.41}$$

then

$$Pipe_c \preceq_P Pipe_a \tag{5.42}$$

Figure 5.3 describes the intuition behind Theorem 5.4.1. A pair of concrete and abstract states are in the simulation relation $\mathcal{S}_P$ if for each parcel state in the concrete pipeline model state, there exists a corresponding abstract parcel state in the abstract pipeline model that simulates it. The parcel commuting diagrams are used to define the abstract pipeline step that makes the diagram commute.

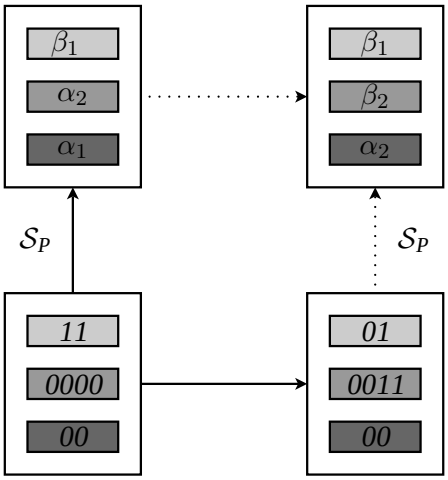


Fig. 5.3a. Pipeline commuting diagram.

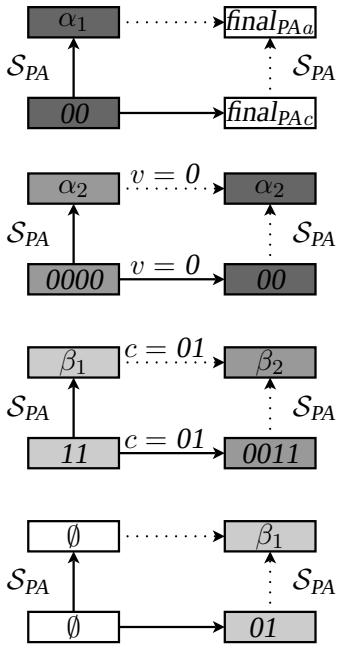


Fig. 5.3b. Parcel commuting diagrams.

Figure 5.3: $AndOr$ example (simulation).

Proof. We define $\mathcal{S}_P$ by the equation

$$\begin{aligned}&(q_{Pc}, q_{Pa}) \in \mathcal{S}_P \equiv \\ &\quad (q_{Pc} =_{V_{ctrl}} q_{Pa}) \wedge (\forall\, q_{PAc} \in \mathit{PclStates}\, q_{Pc}.\ q_{PAc} \preceq_{PA} q_{Pa} \mid_{\mathit{dom}\, q_{PAc}})\end{aligned} \tag{5.43}$$

We need to show $\mathcal{S}_P$ satisfies the commuting diagram in Equation 3.15 and the condition on initial states in Equation 3.16 of Definition 3.2.1.

Commuting Diagram

Consider $(q_{Pc}, q_{Pa}) \in \mathcal{S}_P$ and $(q_{Pc}, t_{Pc}, q'_{Pc}) \in R_{Pc}$. We need to show the following diagram commutes:

$$\begin{array}{ccc} q_{Pa} & \xrightarrow{t_{Pa}} & q_{Pa}{}' \\ \mathcal{S}_P \uparrow & & \uparrow \mathcal{S}_P \\ q_{Pc} & \xrightarrow{t_{Pc}} & q_{Pc}{}' \end{array} \tag{5.44}$$

We apply Theorem 5.3.1 to find $t_{Pa} \in T_{Pa}$ and $q_{Pa}{}' \in Q_{Pa}$ so that the following diagram commutes:

$$\begin{array}{ccc} q_{Pa} & \overset{t_{Pa}}{\cdots\!\rightarrow} & q_{Pa}{}' \\ {\scriptstyle =_P}\uparrow & & \uparrow{\scriptstyle =_P} \\ q_{Pc} & \xrightarrow{t_{Pc}} & q_{Pc}{}' \end{array} \tag{5.45}$$

We then show that

$$(\forall\, q_{PAc}{}' \in \mathit{PclStates}\, q_{Pc}{}'.\ q_{PAc}{}' \preceq_{PA} q_{Pa}{}' \mid_{\mathit{dom}\, q_{PAc}{}'}) \tag{5.46}$$

which implies

$$(q_{Pc}{}', q_{Pa}{}') \in \mathcal{S}_P$$

In order to apply Theorem 5.3.1 we need to provide

$$\begin{aligned} pclTrans_a &: PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc}) \rightarrow T_{PAa} \\ pclNextState_a &: PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc}) \rightarrow Q_{PAa} \end{aligned}$$

so that the following diagram commutes:

$$\begin{array}{l} \forall\, p \in PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc}). \\ \quad \begin{array}{ccc} q_{Pa} \mid_p & \overset{pclTrans_a\ p}{\cdots\!\rightarrow} & pclNextState_a\, p \\ \uparrow & & \uparrow \\ q_{Pc} \mid_p & \xrightarrow{\mathit{pclTrans}\ p} & \mathit{pclNextState}\ p \end{array} \end{array} \tag{5.47}$$

We have

$$\begin{array}{l} \forall\, p \in PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc}). \\ \quad \Big(\ q_{Pc} \mid_p \in \mathit{PclStates}\, q_{Pc} \cup \{\, \emptyset \,\}\ \Big) \wedge \Big(\ (q_{Pc} \mid_p, \mathit{pclTrans}\ p, \mathit{pclNextState}\ p) \in R_{PAc}\ \Big) \end{array} \tag{5.48}$$

Using Equation 5.43 and Equation 5.48 we infer that the following diagram commutes:

$$\forall\, p \in PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc}).$$

$$q_{Pc} \mid_p \neq \emptyset \Longrightarrow \left(\begin{array}{l} \exists\, t_{PAa} \in T_{PAa}. \\ \quad \exists\, {q_{PAa}}' \in Q_{PAa}. \\ \qquad \begin{array}{ccc} q_{Pa} \mid_p & \overset{t_{PAa}}{\cdots\cdots\rightarrow} & {q_{PAa}}' \\ {\scriptstyle =_{PA}}\uparrow & & \uparrow{\scriptstyle =_{PA}} \\ q_{Pc} \mid_p & \xrightarrow{pclTrans\ p} & pclNextState\ p \end{array} \end{array} \right) \tag{5.49}$$

Since $\emptyset \in I_{PAc}$, using Equation 5.41 and Equation 5.48 we infer that the following diagram commutes:

$$\forall\, p \in PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc}).$$

$$q_{Pc} \mid_p = \emptyset \Longrightarrow \left(\begin{array}{l} \exists\, t_{PAa} \in T_{PAa}. \\ \quad \exists\, {q_{PAa}}' \in Q_{PAa}. \\ \qquad \begin{array}{ccc} \emptyset & \overset{t_{PAa}}{\cdots\cdots\rightarrow} & {q_{PAa}}' \\ {\scriptstyle =_{PA}}\uparrow & & \uparrow{\scriptstyle =_{PA}} \\ q_{Pc} \mid_p & \xrightarrow{pclTrans\ p} & pclNextState\ p \end{array} \end{array} \right) \tag{5.50}$$

Since $q_{Pc} \mid_p = \emptyset$ it means that p contains no registers

$$q_{PAa} = q_{Pa} \mid_p = \emptyset \tag{5.51}$$

and therefore we can rewrite Equation 5.50:

$$\forall\, p \in PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc}).$$

$$q_{Pc} \mid_p = \emptyset \Longrightarrow \left(\begin{array}{l} \exists\, t_{PAa} \in T_{PAa}. \\ \quad \exists\, {q_{PAa}}' \in Q_{PAa}. \\ \qquad \begin{array}{ccc} q_{PAa} \mid_p & \overset{t_{PAa}}{\cdots\cdots\rightarrow} & {q_{PAa}}' \\ {\scriptstyle =_{PA}}\uparrow & & \uparrow{\scriptstyle =_{PA}} \\ q_{Pc} \mid_p & \xrightarrow{pclTrans\ p} & pclNextState\ p \end{array} \end{array} \right) \tag{5.52}$$

Combining Equation 5.49 and Equation 5.52 we get:

$$\begin{array}{l} \forall\, p \in PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc}). \\ \quad \exists\, t_{PAa} \in T_{PAa}.\ \exists\, {q_{PAa}}' \in Q_{PAa}. \\ \qquad \begin{array}{ccc} q_{Pa} \mid_{\,p} & \overset{t_{PAa}}{\cdots\cdots\rightarrow} & {q_{PAa}}' \\ \uparrow & & \vdots \\ q_{Pc} \mid_{\,p} & \overset{pclTrans\ p}{\longrightarrow} & pclNextState\ p \end{array} \end{array} \tag{5.53}$$

Using Equation 5.53 we can define $pclTrans_a$ and $pclNextState_a$ so that Equation 5.47 holds.

Initial States

Let $q_{Pc} \in I_{Pc}$. We show there exists $q_{Pa} \in I_{Pa}$ so that $q_{Pc} \preceq_P q_{Pa}$.

On control variables, we must define q_{Pa} so that the following equation holds:

$$q_{Pc} =_{V_{ctrl}} q_{Pa} \tag{5.54}$$

Since $Pipe_a$ is an abstract interpretation of $Pipe_c$, in the two models the initial conditions with respect to control variables are the same.

On parcel variables, we must define q_{Pa} so that

$$\forall\, q_{PAc} \in PclStates\, q_{Pc}.\ q_{PAc} \preceq_{PA} q_{Pa} \mid_{dom\, q_{PAc}} \tag{5.55}$$

Since $q_{Pc} \in I_{Pc}$ we have $PclStates\, q_{Pc} \subseteq I_{PAc}$. Therefore

$$\forall\, q_{PAc} \in PclStates\, q_{Pc}.\ \exists\, q_{PAa} \in I_{PAa}.\ q_{PAc} \preceq_{PA} q_{PAa}$$

Denoting the Skolem constant by $f_\exists$, the previous equation expresses as

$$\forall\, q_{PAc} \in PclStates\, q_{Pc}.\ q_{PAc} \preceq_{PA} f_\exists(q_{PAc})$$

Due to the condition that parcel states in initial states are fixed, as stated in Equation 5.14, $PclStates\, q_{Pc}$ is a partition of q_{Pc}. We can therefore define q_{Pa} by the following equation:

$$\forall\, q_{PAc} \in PclStates\, q_{Pc}.\ q_{Pa} \mid_{dom\, q_{PAc}} = f_\exists(q_{PAc}) \tag{5.56}$$

Given that all abstract interpretations satisfy Equation 5.12, using Equation 5.56 we infer that

$$q_{Pa} \models \mathit{Init}_{pcl\,a} \tag{5.57}$$

Equation 5.54 and Equation 5.57 imply that $q_{Pa} \models Pipe_{\,a}.\mathit{Init}$. □

5.4.2 Language Containment

We say parcels duplicate if two distinct parcels at the current step are continuations of the same parcel at the previous step.

$$\begin{array}{l}\exists\, q_P.\ \exists\, t_P.\ \exists\, {q_P}'.\ \exists\, {t_P}'.\ \exists\, {q_P}''.\\ \quad (q_P, t_P, q'_P) \in R_P \wedge (q'_P, t'_P, q''_P) \in R_P \wedge \\ \quad\left(\begin{array}{l}\exists\, p_1 \in PclMap\ (q_P, t_P, q'_P).\ \exists\, p_2 \in PclMap\ (q'_P, t'_P, q''_P).\ \exists\, p_3 \in PclMap\ (q'_P, t'_P, q''_P).\\ \quad p_2 \neq p_3 \wedge \mathit{pclState}\ p_2 \subseteq \mathit{pclNextState}\ p_1 \wedge \mathit{pclState}\ p_3 \subseteq \mathit{pclNextState}\ p_1\end{array}\right)\end{array} \tag{5.58}$$

Equation 5.58 is reformulated equivalently to Equation 5.59.

$$\begin{array}{l}\exists\, q_P.\ \exists\, t_P.\ \exists\, {q_P}'.\ \exists\, {t_P}'.\ \exists\, {q_P}''.\\ \quad (q_P, t_P, q'_P) \in R_P \wedge (q'_P, t'_P, q''_P) \in R_P \wedge \\ \quad\left(\begin{array}{l}\exists\, v_1 \in \mathit{RegPcl}.\ \exists\, v_2 \in \mathit{RegPcl}.\ \exists\, p_1 \in PclMap\ (q_P, t_P, q'_P).\\ \quad (PclMap_{\,alt}\ (q'_P, t'_P, q''_P)\ v_1 \neq PclMap_{\,alt}\ (q'_P, t'_P, q''_P)\ v_2) \wedge {v_1}' \in p_1^* \wedge {v_2}' \in p_1^*\end{array}\right)\end{array} \tag{5.59}$$

Equation 5.59 holds if and only if the following propositional formula is satisfiable:

$$\left(\begin{array}{c}[\![R_P]\!]_{bool}\,({V_{step}}^1) \wedge [\![PclMap_{\,alt}]\!]_{bool}\,({V_{step}}^1, {V_{pclMap}}^1) \wedge FanOut({V_{step}}^1, {V_{pclMap}}^1, {V_{fanOut}}^1)\\ \wedge\\ [\![R_P]\!]_{bool}\,({V_{step}}^2) \wedge [\![PclMap_{\,alt}]\!]_{bool}\,({V_{step}}^2, {V_{pclMap}}^2)\\ \wedge\\ \bigwedge_{v \in V_{reg}} ((v^1)' = v^2)\\ \wedge\\ \bigvee_{v_1^2 \in \mathit{RegPcl}^2}\ \bigvee_{v_2^2 \in \mathit{RegPcl}^2} (pclMap^2_{v_1^2} \neq pclMap^2_{v_2^2} \wedge fanOut^1_{(v_1^1)'} = fanOut^1_{(v_2^1)'})\end{array}\right) \tag{5.60}$$

We can therefore verify parcel separation using a Boolean solver.

We formalize the parcel automaton runs that occur within a pipeline model run $\sigma_P \in \mathcal{L}(Pipe)$. In order to find the parcels that belong to the same run we define the partial order '$\underset{\sigma_P}{\ll}$' on the parcels of the pipeline run so that '$\underset{\sigma_P}{\ll}$' corresponds to the transitive closure of the relationship of a parcel continuing another. When parcels do not duplicate, every parcel in the pipeline run belongs to a unique maximal chain (totally ordered subset) of '$\underset{\sigma_P}{\ll}$'. Each such maximal chain in turn corresponds to a run of the parcel automaton.

5.4.2 Definition (Parcel Order)**.** We define the relation $\underset{\sigma_P}{<}$ over the set $(\mathcal{P}(V_{pcl}) \setminus \emptyset) \times \mathbf{N}$:

$$\begin{aligned}(p^n, n) \underset{\sigma_P}{<} (p^{n+1}, n+1) \equiv \\ p^n \in PclMap\ (q_P^n, t_P^n, q_P^{n+1}) \wedge p^{n+1} \in PclMap\ (q_P^{n+1}, t_P^{n+1}, q_P^{n+2}) \wedge \\ \mathit{pclState}\ p^{n+1} \subseteq \mathit{pclNextState}\ p^n\end{aligned} \tag{5.61}$$

The relation '$\underset{\sigma_P}{\ll}$' is the reflexive and transitive closure of '$\underset{\sigma_P}{<}$'.

The relation '$\underset{\sigma_P}{\ll}$' is by definition transitive and reflexive. To show the relation is also antisymmetric, consider

$$(p^{n_1}, n_1) \underset{\sigma_P}{\ll} (p^{n_2}, n_2) \tag{5.62}$$

$$(p^{n_2}, n_2) \underset{\sigma_P}{\ll} (p^{n_1}, n_1) \tag{5.63}$$

We must have $n_1 \leq n_2$ and $n_2 \leq n_1$ and so $n_1 = n_2$. The only possibility is to also have $p^{n_1} = p^{n_2}$.

Given the run in Figure 5.4 of the $AndOr$ example, first described in Section 4.1, we have:

$$\begin{aligned}(\{\, r_1 \,\}, 0) &\underset{\sigma_P}{<} (\{\, r_2 \,\}, 1) \\ (\{\, r_2 \,\}, 0) &\underset{\sigma_P}{<} (\{\, r_3 \,\}, 1) \\ (\{\, v_i \,\}, 0) &\underset{\sigma_P}{<} (\{\, r_1 \,\}, 1) \\ (\{\, v_i \,\}, 0) &\underset{\sigma_P}{\ll} (\{\, r_2 \,\}, 2)\end{aligned}$$

Let $(p^n, n) \in (\mathcal{P}(V_{pcl}) \setminus \emptyset) \times \mathbf{N}$. We make the following two observations.

1. If parcels do not duplicate then (p^n, n) has at most one successor in Equation 5.61.
2. The third property of parcel maps ensures that (p^n, n) has at most one predecessor in Equation 5.61.

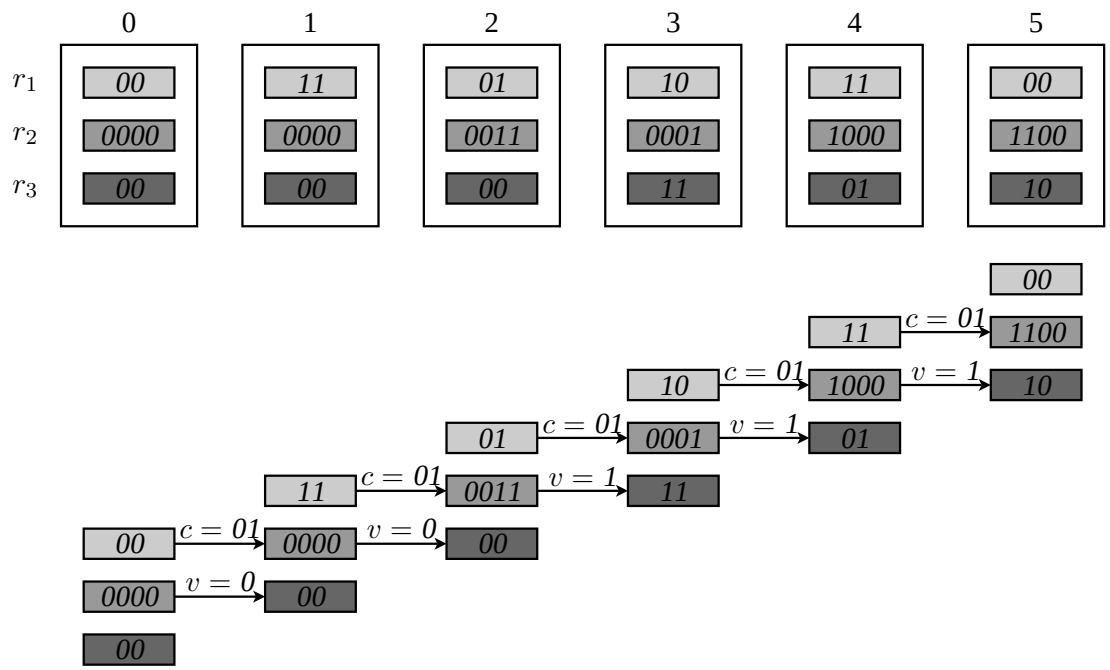


Figure 5.4: $AndOr$ Computation.

5.4.3 Definition (Chain)**.** $C \subseteq \mathcal{P}((\mathcal{P}(V_{pcl}) \setminus \emptyset) \times \mathbf{N})$ is a chain of $\underset{\sigma_P}{\ll}$ if any two elements are comparable:

$$\forall\,(p^{n_1}, n_1) \in C.\, \forall\,(p^{n_2}, n_2) \in C.\, (p^{n_1}, n_1) \underset{\sigma_P}{\ll} (p^{n_2}, n_2) \vee (p^{n_2}, n_2) \underset{\sigma_P}{\ll} (p^{n_1}, n_1) \qquad (5.64)$$

A chain C is maximal if it does not occur as a strict subset of another chain.

For our example in Figure 5.4, some examples of maximal chains are:

$$\begin{array}{llllll}
(\{\, r_1 \,\}, 0) & \underset{\sigma_P}{<} & (\{\, r_2 \,\}, 1) & \underset{\sigma_P}{<} & (\{\, r_3 \,\}, 2) & \\
(\{\, r_2 \,\}, 0) & \underset{\sigma_P}{<} & (\{\, r_3 \,\}, 1) & & & \\
(\{\, v_i \,\}, 0) & \underset{\sigma_P}{<} & (\{\, r_1 \,\}, 1) & \underset{\sigma_P}{<} & (\{\, r_2 \,\}, 2) & \underset{\sigma_P}{<} \quad (\{\, r_3 \,\}, 3)
\end{array}$$

5.4.4 Proposition. If parcels do not duplicate then every element of the set $(\mathcal{P}(V_{pcl}) \setminus \emptyset) \times \mathbf{N}$ belongs to a unique maximal chain of '$\underset{\sigma_P}{\ll}$'. For $(p^n, n) \in (\mathcal{P}(V_{pcl}) \setminus \emptyset) \times \mathbf{N}$ we denote the corresponding maximal chain by *chain* (p^n, n).

Proof. **Existence** We define:

$$\begin{aligned} \textit{chain}\,(p^n, n) \equiv & \{\, (p^{n-k}, n-k) \mid (p^{n-k}, n-k) \underset{\sigma_P}{\ll} (p^n, n) \,\} \\ & \cup \{\, (p^{n+k}, n+k) \mid (p^n, n) \underset{\sigma_P}{\ll} (p^{n+k}, n+k) \,\} \end{aligned} \tag{5.65}$$

We need to show *chain* (p^n, n) is a chain. Consider two distinct elements

$$\begin{aligned} (p^{n_1}, n_1) &\in \textit{chain}\,(p^n, n) \\ (p^{n_2}, n_2) &\in \textit{chain}\,(p^n, n) \end{aligned}$$

We have to show that either of the following holds:

$$\begin{aligned} (p^{n_1}, n_1) &\underset{\sigma_P}{\ll} (p^{n_2}, n_2) \\ (p^{n_2}, n_2) &\underset{\sigma_P}{\ll} (p^{n_1}, n_1) \end{aligned}$$

If

$$\begin{aligned} (p^{n_1}, n_1) &\underset{\sigma_P}{\ll} (p^n, n) \\ (p^n, n) &\underset{\sigma_P}{\ll} (p^{n_2}, n_2) \end{aligned}$$

or

$$\begin{aligned} (p^{n_2}, n_2) &\underset{\sigma_P}{\ll} (p^n, n) \\ (p^n, n) &\underset{\sigma_P}{\ll} (p^{n_1}, n_1) \end{aligned}$$

by transitivity we get the desired conclusion. Consider the case when

$$\begin{aligned} (p^{n_1}, n_1) &\underset{\sigma_P}{\ll} (p^n, n) \\ (p^{n_2}, n_2) &\underset{\sigma_P}{\ll} (p^n, n) \end{aligned}$$

Therefore, we must have:

$$\begin{array}{rcccccccl} (p^{n_1}, n_1) & = & a_0 & \underset{\sigma_P}{<} & \cdots & \underset{\sigma_P}{<} & a_{i-1} & \underset{\sigma_P}{<} & a_i & = & (p^n, n) \\ (p^{n_2}, n_2) & = & b_0 & \underset{\sigma_P}{<} & \cdots & \underset{\sigma_P}{<} & b_{j-1} & \underset{\sigma_P}{<} & b_j & = & (p^n, n) \end{array}$$

We show that either $a_0 \in \{ b_0, \ldots, b_j \}$ or $b_0 \in \{ a_0, \ldots, a_i \}$. Assume by contradiction that that is not the case. We consider the smallest $k \in \{ 1, \ldots, i \}$ so that $a_k = b_l$ for some $l \in \{ 1, \ldots, j \}$. We have

$$\begin{array}{rcl} a_{k-1} & \underset{\sigma_P}{<} & a_k \\ b_{l-1} & \underset{\sigma_P}{<} & a_k \end{array}$$

It follows that a_k has two distinct predecessors, contradicting our first previous observation. The case when

$$\begin{array}{rcl} (p^n, n) & \underset{\sigma_P}{\ll} & (p^{n_1}, n_1) \\ (p^n, n) & \underset{\sigma_P}{\ll} & (p^{n_2}, n_2) \end{array}$$

is treated similarly, arriving to a contradiction of our second observation.

Uniqueness If C is a chain containing (p^n, n) then according to Equation 5.65 we must have $C \subseteq \mathit{chain}\,(p^n, n)$. Since C is maximal we must have $C = \mathit{chain}\,(p^n, n)$. □

For our example, the set of maximal chains to which every element (p^n, n) belongs to is described as follows:

$$\begin{array}{lcllcl} (\{ r_1 \}, 0) & \underset{\sigma_P}{<} & (\{ r_2 \}, 1) & \underset{\sigma_P}{<} & (\{ r_3 \}, 2) & \\ (\{ r_2 \}, 0) & \underset{\sigma_P}{<} & (\{ r_3 \}, 1) & & & \\ (\{ r_3 \}, 0) & & & & & \\ (\{ v_i \}, n) & \underset{\sigma_P}{<} & (\{ r_1 \}, n+1) & \underset{\sigma_P}{<} & (\{ r_2 \}, n+2) & \underset{\sigma_P}{<} \; (\{ r_3 \}, n+3)\,,\; n \geq 0 \end{array}$$

5.4.5 Proposition. If $C \subseteq \mathcal{P}((\mathcal{P}(V_{pcl}) \setminus \emptyset) \times \mathbf{N})$ is a maximal chain of $\underset{\sigma_P}{\ll}$ then:

- If C is finite then C has form

$$C = \{ (p^n, n), \ldots, (p^{n+k}, n+k) \} \tag{5.66}$$

and

$$\forall\, j \in \{ n, \ldots, n+k-1 \}.\, (p^j, j) \underset{\sigma_P}{<} (p^{j+1}, j+1) \tag{5.67}$$

- If C is infinite then C has form

$$C = \{\, (p^{n+k}, n+k) \mid k \in \mathbf{N} \,\} \tag{5.68}$$

and

$$\forall\, k \in \mathbf{N}.\ (p^{n+k}, n+k) \underset{\sigma_P}{<} (p^{n+k+1}, n+k+1) \tag{5.69}$$

Proof. Since any two elements are comparable by $\underset{\sigma_P}{\ll}$ and since the definition of $\underset{\sigma_P}{<}$ implies that if $(p^{n_1}, n_1) \underset{\sigma_P}{<} (p^{n_2}, n_2)$ then $n_1 < n_2$ then C must have form:

$$(p^n, n) \underset{\sigma_P}{\ll} (p^{n+k_1}, n+k_1) \underset{\sigma_P}{\ll} (p^{n+k_2}, n+k_2) \underset{\sigma_P}{\ll} \cdots$$

where

$$k_1 < k_2 \cdots$$

Since C is maximal we must have

$$\begin{aligned} k_1 &= 1 \\ k_2 &= 2 \\ &\vdots \end{aligned}$$

□

5.4.6 Definition (Associated Parcel Automaton Run)**.** Let

$$C = (p^n, n) \underset{\sigma_P}{<} (p^{n+1}, n+1) \underset{\sigma_P}{<} \cdots$$

We define

$$\mathit{run}_{PA}\ C : \mathbf{N} \to Q_{PA} \times T_{PA}$$

- C is infinite

$$\mathit{run}_{PA}\ C\ j \equiv (q_P^{n+j} \mid_{p^{n+j}}, \mathit{pclTrans}\ p^{n+j}) \tag{5.70}$$

- $C = (q_{PA}^n, n) \underset{\sigma_P}{<} \cdots \underset{\sigma_P}{<} (q_{PA}^{n+k}, n+k)$

$$\mathit{run}_{PA}\ C\ j \equiv \begin{cases} (q_P^{n+j} \mid_{p^{n+j}}, \mathit{pclTrans}\ p^{n+j}) \ :\ j \leq k \\ (\mathit{final}_{PA}, \emptyset) \ :\ j > k \end{cases} \tag{5.71}$$

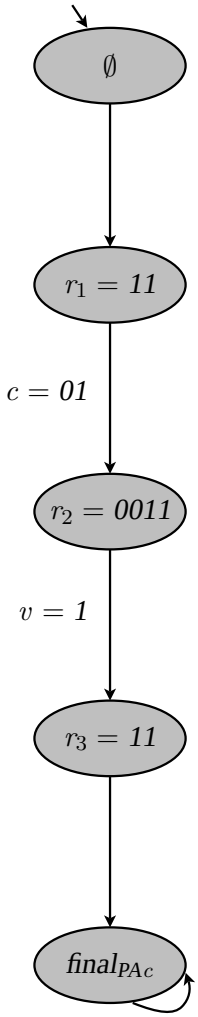


Figure 5.5: Associated parcel automaton run.

The associated run for the chain

$$(\{\, v_i \,\}, 0) \underset{\sigma_P}{<} (\{\, r_1 \,\}, 1) \underset{\sigma_P}{<} (\{\, r_2 \,\}, 2) \underset{\sigma_P}{<} (\{\, r_3 \,\}, 3)$$

is shown in Figure 5.5.

5.4.7 Proposition. If $C = (p^n, n) \underset{\sigma_P}{<} (p^{n+1}, n+1) \underset{\sigma_P}{<} \cdots$ is a maximal chain then $run_{PA}\ C \in \mathcal{L}(pa(Pipe, PclMap))$.

Proof. Let $\mathit{run}_{PA}\ C\ j = (q^j_{PA}, t^j_{PA})$. We need to show

$$(q^j_{PA}, t^j_{PA}, q^{j+1}_{PA}) \in R_{PA} \tag{5.72}$$

$$q^0_{PA} \in I_{PA} \tag{5.73}$$

If C is infinite, $\mathit{run}_{PA}\ C$ is defined by Equation 5.70. Therefore, Equation 5.72 is equivalent to

$$\left(\ q_P^{n+j} \mid_{\ p^{n+j}}, \mathit{pclTrans}\ p^{n+j}, q_P^{n+j+1} \mid_{\ p^{n+j+1}} \ \right) \in R_{PA}$$

which holds because $(p^{n+j}, n+j) \underset{\sigma_P}{<} (p^{n+j+1}, n+j+1)$.

If $C = (q^n_{PA}, n) \underset{\sigma_P}{<} \cdots \underset{\sigma_P}{<} (q^{n+k}_{PA}, n+k)$ is finite then $\mathit{run}_{PA}\ C$ is defined according to Equation 5.71.

- $j < k$ Equation 5.72 is equivalent to

  $$\left(\ q_P^{n+j} \mid_{\ p^{n+j}}, \mathit{pclTrans}\ p^{n+j}, q_P^{n+j+1} \mid_{\ p^{n+j+1}} \ \right) \in R_{PA}$$

  which holds because $(p^{n+j}, n+j) \underset{\sigma_P}{<} (p^{n+j+1}, n+j+1)$.

- $j = k$ Equation 5.72 is equivalent to

  $$\left(\ q_P^{n+j} \mid_{\ p^{n+j}}, \mathit{pclTrans}\ p^{n+j}, \mathit{final}_{PA} \ \right) \in R_{PA}$$

  which holds because any parcel automaton state can transition to the final state.

- $j > k$ Equation 5.72 is equivalent to

  $$(\mathit{final}_{PA}, \emptyset, \mathit{final}_{PA}) \in R_{PA}$$

  which holds according to the definition of parcel automata.

Equation 5.73 is equivalent to

$$q_P^n \mid_{\ p^n} \in I_{PA} \tag{5.74}$$

We consider two cases:

- $n = 0$. In this case $q^0_P \in I_P$ and therefore $q_P^0 \mid_{\ p^0} \in I_{PA}$ according to the definition of the induced parcel automaton.

- $n > 0$. We consider two subcases:

  - $p^n \cap \mathit{RegPcl} = \emptyset$ It follows that $q_P^n \mid_{\ p^n} = \emptyset$. Since $\emptyset \in I_{PA}$, Equation 5.74 holds.

– $p^n \cap \mathit{RegPcl} \neq \emptyset$ This leads to a contradiction since p^n must continue a parcel p^{n-1} in the previous step $(q_P^{n-1}, t_P^{n-1}, q_P^n)$. We therefore have $(p^{n-1}, n-1) \underset{\sigma_P}{<} (p^n, n)$ which contradicts the fact that C is maximal since $C \cup \{\, (p^{n-1}, n-1) \,\}$ is a chain.

□

5.4.8 Theorem (Abstraction Using Language Containment)**.** Let $Pipe_c$ and $Pipe_a$ be two pipeline models such that

$$Pipe_c = Pipe_a \left[Dps_a / Dps_c \right] \tag{5.75}$$

If the following conditions are met

1. Parcels do not duplicate in $Pipe_c$.
2. Language containment of parcel automata.

$$\mathcal{L}(\mathit{pa}(Pipe_c, PclMap)) \subseteq_{PA} \mathcal{L}(pa_a) \tag{5.76}$$

then language containment of pipeline model holds:

$$\mathcal{L}(Pipe_c) \subseteq_P \mathcal{L}(Pipe_a) \tag{5.77}$$

Figure 5.6 describes the construction we use in the proof of Theorem 5.4.8. We first identify the maximal chains of $\underset{\sigma_P}{\ll}$, then, corresponding to each chain we have a concrete parcel automaton run. For each such run there exists an equivalent abstract run. The abstract parcel runs are used to create the abstract pipeline model run.

Proof. Let $\sigma_{Pc} \in \mathcal{L}(Pipe_c)$.

We need to show there exists $\sigma_{Pa} \in \mathcal{L}(Pipe_a)$ such that

$$\sigma_{Pc} =_P \sigma_{Pa} \tag{5.78}$$

We will define $\sigma_{Pa} : \mathbf{N} \to Q_{Pa} \times T_{Pa}$ by induction. We recall the notation

$$\begin{aligned} \sigma_{Pc}\, n &= (q_{Pc}^n, t_{Pc}^n) \\ \sigma_{Pa}\, n &= (q_{Pa}^n, t_{Pa}^n) \end{aligned}$$

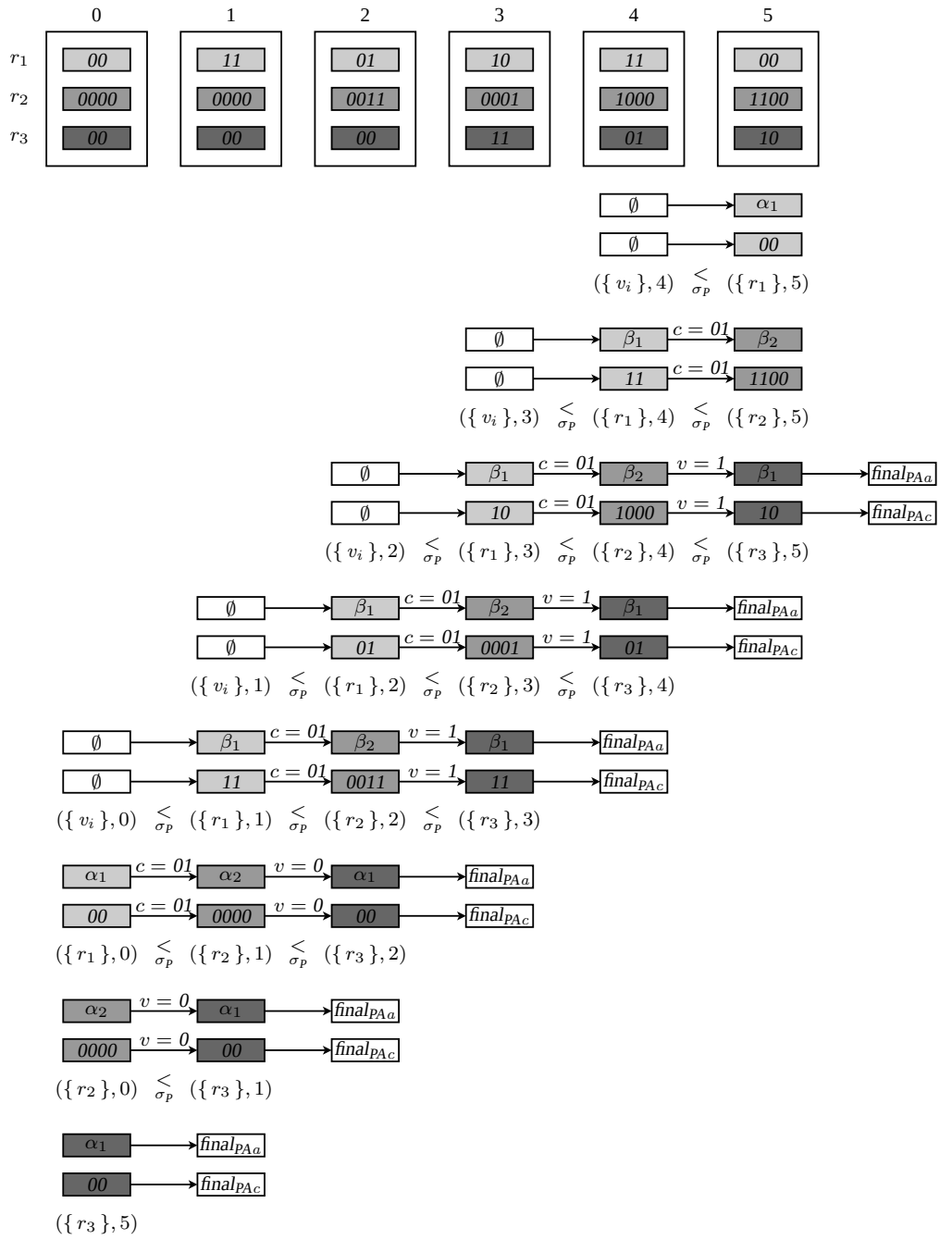


Figure 5.6: Construction in Theorem 5.4.8.

For $n \geq 0$ we will be applying Theorem 5.3.1 to q_{Pa}^n and derive $t_{Pa}^n \in T_{Pa}$ and $q_{Pa}^{n+1} \in Q_{Pa}$ such

that the diagram commutes:

$$\begin{array}{ccc} q^n_{Pa} & \xrightarrow{t^n_{Pa}} & q^{n+1}_{Pa} \\ {\scriptstyle =_P}\uparrow & & \uparrow{\scriptstyle =_P} \\ q^n_{Pc} & \xrightarrow{t^n_{Pc}} & q^{n+1}_{Pc} \end{array} \tag{5.79}$$

If Equation 5.79 holds inductively and $q^0_{Pa} \in I_{Pa}$ then $\sigma_{Pa} \in \mathcal{L}(Pipe_a)$ and Equation 5.78 holds.

For each $n \geq 0$ and $p^n \in PclMap\,(q^n_{Pc}, t^n_{Pc}, q^{n+1}_{Pc})$ we denote the parcel automaton run associated with *chain* p^n by $\sigma_{PAc\,p^n}$:

$$\sigma_{PAc\,p^n} \equiv \mathit{run}_{PA}\,(\mathit{chain}\; p^n)$$

According to Definition 5.4.6, if (p^n, n) occurs on position j_n in its chain then

$$(\sigma_{PAc\,p^n})(j_n) = (q^n_{Pc} \mid_{p^n}, \mathit{pclTrans}\; p^n) \tag{5.80}$$

Since $\mathcal{L}(\mathit{pa}(Pipe_c, PclMap)) \subseteq_P \mathcal{L}(pa_a)$ there exists $\sigma_{PAa\,p^n} \in \mathcal{L}(\mathit{pa}(Pipe_a))$ so that

$$\sigma_{PAc\,p^n} =_{PA} \sigma_{PAa\,p^n} \tag{5.81}$$

We use the notation:

$$\begin{array}{rcl} (\sigma_{PAc\,p^n})(k) & = & (q^k_{PAc\,p^n}, t^k_{PAc\,p^n}) \\ (\sigma_{PAa\,p^n})(k) & = & (q^k_{PAa\,p^n}, t^k_{PAa\,p^n}) \end{array}$$

Equation 5.81 implies that the the following diagram commutes:

$$\begin{array}{ccc} q^k_{PAa\,p^n} & \xrightarrow{t^k_{PAa\,p^n}} & q^{k+1}_{PAa\,p^n} \\ {\scriptstyle =_{PA}}\uparrow & & \uparrow{\scriptstyle =_{PA}} \\ q^k_{PAc\,p^n} & \xrightarrow{t^k_{PAc\,p^n}} & q^{k+1}_{PAc\,p^n} \end{array} \tag{5.82}$$

For $k = j_n$ we have:

$$\begin{array}{rcl} q^{j_n}_{PAc\,p^n} & = & q^n_{Pc} \mid_{p^n} \\ t^{j_n}_{PAc\,p^n} & = & \mathit{pclTrans}\; p^n \\ q^{j_n+1}_{PAc\,p^n} & = & \mathit{pclNextState}\; p^n \end{array}$$

The diagram in Equation 5.82 becomes

$$\begin{array}{ccc} q^{j_n}_{PAa\ p^n} & \overset{t^{j_n}_{PAa\ p^n}}{\cdots\cdots\rightarrow} & q^{j_n+1}_{PAa\ p^n} \\ {\scriptstyle =_{PA}}\uparrow & & \uparrow{\scriptstyle =_{PA}} \\ q^n_{Pc} \mid_{p^n} & \xrightarrow{pclTrans\ p^n} & pclNextState\ p^n \end{array} \tag{5.83}$$

We will define q^n_{PAa} so that:

$$q^n_{PAa} \mid_{p^n} = q^{j_n}_{PAa\ p^n} \tag{5.84}$$

Base case

On control variables we must have:

$$q^0_{Pa} =_{V_{ctrl}} q^0_{Pc} \tag{5.85}$$

On parcel variables q^0_{Pa} is defined according to Equation 5.84:

$$\forall\, p^0 \in PclMap\ (q^0_{Pc}, t^0_{Pc}, q^1_{Pc}).\ q^0_{Pa} \mid_{p^0} = q^{j_0}_{PAa\ p^0} \tag{5.86}$$

Since parcel automaton runs begin at index 0 we have

$$\forall\, p^0 \in PclMap\ (q^0_{Pc}, t^0_{Pc}, q^1_{Pc}).\ q^{j_0}_{PAa\ p^0} = q^0_{PAa\ p^0} \wedge q^0_{PAa\ p^0} \in I_{PAa} \tag{5.87}$$

Since abstract interpretations satisfy Equation 5.12, using Equation 5.86 and Equation 5.87 we infer that

$$q^0_{Pa} \models Init_{pcl\ a} \tag{5.88}$$

Equation 5.85 and Equation 5.88 imply that $q^0_{Pa} \models Pipe_a.Init$.

Inductive case

We assume the following inductive hypothesis:

$$\forall\, p^n \in PclMap\ (q^n_{Pc}, t^n_{Pc}, q^{n+1}_{Pc}).\ q^n_{Pa} \mid_{p^n} \quad = \quad q^{j_n}_{PAa\ p^n} \tag{5.89}$$

$$q^n_{Pa} \quad =_{V_{ctrl}} \quad q^n_{Pc} \tag{5.90}$$

In order to apply Theorem 5.3.1 we need to define

$$pclTrans_a : PclMap\ (q^n_{Pc}, t^n_{Pc}, q^{n+1}_{Pc}) \rightarrow T_{PAa}$$

$$pclNextState_a : PclMap\ (q^n_{Pc}, t^n_{Pc}, q^{n+1}_{Pc}) \rightarrow Q_{PAa}$$

so that the following diagram commutes:

$$\begin{array}{l} \forall\ p^n \in PclMap\ (q^n_{Pc}, t^n_{Pc}, q^{n+1}_{Pc}). \\ \quad q^n_{Pa} \mid {}_{p^n} \overset{pclTrans_a\ p^n}{\cdots\cdots\rightarrow} pclNextState_a\, p^n \\ \qquad\qquad \uparrow \qquad\qquad\qquad\qquad \vdots \\ \quad q^n_{Pc} \mid {}_{p^n} \overset{pclTrans\ p^n}{\longrightarrow} pclNextState\ p^n \end{array} \tag{5.91}$$

Using the inductive hypothesis, the diagram in Equation 5.91 becomes:

$$\begin{array}{l} \forall\ p^n \in PclMap\ (q^n_{Pc}, t^n_{Pc}, q^{n+1}_{Pc}). \\ \quad q^{j_n}_{PAa\ p^n} \overset{pclTrans_a\ p^n}{\cdots\cdots\rightarrow} pclNextState_a\, p^n \\ \qquad\qquad \uparrow \qquad\qquad\qquad\qquad \vdots \\ \quad q^n_{Pc} \mid {}_{p^n} \overset{pclTrans\ p^n}{\longrightarrow} pclNextState\ p^n \end{array} \tag{5.92}$$

We define $pclTrans_a$ and $pclNextState_a$ as follows:

$$pclTrans_a\ p^n = t^{j_n}_{PAa\ p^n} \tag{5.93}$$

$$pclNextState_a\ p^n = q^{j_n+1}_{PAa\ p^n} \tag{5.94}$$

With this definition, the diagram in Equation 5.92 commutes because it is identical to the one in Equation 5.83.

Applying Theorem 5.3.1 we obtain $t^n_{Pa} \in T_{Pa}$ and $q^{n+1}_{Pa} \in Q_{Pa}$ so that the diagram in Equation 5.79 commutes. It remains to show that we maintain our inductive hypothesis:

$$\forall\ p^{n+1} \in PclMap\ (q^{n+1}_{Pc}, t^{n+1}_{Pc}, q^{n+1+1}_{Pc}).\ q^{n+1}_{Pa} \mid {}_{p^{n+1}} = q^{j_{n+1}}_{PAa\ p^{n+1}} \tag{5.95}$$

Parcels at step $(q^{n+1}_{Pc}, t^{n+1}_{Pc}, q^{n+2}_{Pc})$ are either combinational or continue continue a parcel at the previous step. If p^{n+1} is combinational then

$$q^{n+1}_{Pa} \mid {}_{p^{n+1}} = \emptyset$$

$$q^{j_{n+1}}_{PAa\ p^{n+1}} = \emptyset$$

and therefore Equation 5.95 holds.

One of the postconditions of Theorem 5.3.1 (Equation 5.22) implies the following:

$$\begin{aligned} &\forall\, p^{n+1} \in PclMap\ (q_{Pc}^{n+1}, t_{Pc}^{n+1}, q_{Pc}^{n+2}). \\ &\quad p^{n+1} \cap \textit{RegPcl} \neq \emptyset \Longrightarrow \\ &\qquad \exists\, p^{n} \in PclMap\ (q_{Pc}^{n}, t_{Pc}^{n}, q_{Pc}^{n+1}).\ q_{Pa}^{n+1} \mid_{\ p^{n+1}} = q_{PAa\ p^{n}}^{j_n+1} \end{aligned} \tag{5.96}$$

If $p^{n+1} \cap \textit{RegPcl} \neq \emptyset$ then there exists $p^n \in PclMap\ (q_{Pc}^{n+1}, t_{Pc}^{n+1}, q_{Pc}^{n+1+1})$ so that

$$p^n \underset{\sigma_{Pc}}{<} p^{n+1}$$

which implies that that the two parcels belong to the same run of $\textit{pa}(Pipe_a)$ and therefore

$$q_{PAa\ p^{n+1}}^{j_{n+1}} = q_{PAa\ p^{n}}^{j_n+1} \tag{5.97}$$

Equation 5.96 and Equation 5.97 imply that Equation 5.95 holds $p^{n+1} \cap \textit{RegPcl} \neq \emptyset$.

□

5.5 Summary

Parcel independence is used to prove Theorem 5.3.1 that states that commuting diagrams between the concrete and abstract parcel automaton states imply a commuting diagram between the containing concrete and abstract pipeline states. Theorem 5.3.1 is used to prove soundness of abstraction using parcel automata for simulation in Theorem 5.4.1 and respectively, for language containment in Theorem 5.4.8.

Chapter 6

Abstraction Of Parcel Automata

This chapter describes an algorithm that abstracts the parcel automaton $pa_c = \mathit{pa}(Pipe_c, PclMap)$ induced by the parcel map to a parcel automaton pa_a such that

$$pa_c \preceq_{PA} pa_a \tag{6.1}$$

$$\mathcal{L}(pa_c) \subseteq_{PA} \mathcal{L}(pa_a) \tag{6.2}$$

The abstract parcel automaton pa_a is used to produce an abstract interpretation of the pipeline datapath, resulting in a pipeline model $Pipe_a$ such that:

$$Pipe_a = Pipe_c \left[Dps_a / Dps_c \right] \tag{6.3}$$

Applying Theorem 5.4.1 to Equation 6.1 and Theorem 5.4.8 to Equation 6.2 we get:

$$Pipe_c \preceq_P Pipe_a$$

$$\mathcal{L}(Pipe_c) \subseteq_P Pipe_a$$

Thus the datapath abstraction algorithm preserves control properties.

Section 6.1 formalizes path abstraction and proves that it implies simulation between parcel automata. In Section 6.2 we define inductively the parcel automaton $pa_{c\,1}$ that is an an approximation of the induced parcel automaton. The parcel automaton $pa_{c\,1}$ may also represent unreachable datapath behaviours but is more practical to represent than the induced parcel automaton. Path abstraction is based on the systematic exploration of the paths through the parcel automaton $pa_{c\,1}$. For each such path there is an equivalent one in the the abstract parcel automaton.

The encoding of the approximate parcel automaton in propositional logic is presented in Section 6.3

and Section 6.4. An abstraction algorithm that is based on path abstraction of the approximate parcel automaton is presented in Section 6.5. We present experimental results in Section 6.6.

6.1 Path Abstraction

In this section, an abstraction that maps concrete paths to abstract states is shown to be conservative, i.e. implies simulation, in Lemma 6.1.1. The notion of paths that are not distinguishable by the control is captured using path equivalence. Finite paths and infinite runs are connected through the concept of terminating run, which requires that a run consist of a finite prefix in which all state updating datapath computations are confined, followed by an infinite suffix that contains only value copying transitions. Lemma 6.1.4 gives sufficient conditions under which path abstraction preserves language containment.

Given a parcel automaton run $\sigma_{PA} \in \mathcal{L}(pa)$, we use the following notation for the finite path from state q^0_{PA} to state q^k_{PA} that occurs in the prefix of length k of σ_{PA}:

$$\pi^k_{PA} = q^0_{PA} \xrightarrow{t^0_{PA}} \cdots \xrightarrow{t^{k-1}_{PA}} q^k_{PA} \tag{6.4}$$

When $k = 0$ the path π^k_{PA} reduces to q^0_{PA}. The set of all such paths is denoted by $\Pi(pa)$:

$$\Pi(pa) \equiv \{\, \pi^k_{PA} \mid k \in \mathbf{N} \wedge \exists\, \sigma_{PA} \in \mathcal{L}(pa).\ \pi^k_{PA} \text{ is a prefix of } \sigma_{PA} \,\}$$

Given the path π^k_{PA} in Equation 6.4 we use the notation:

$$\begin{array}{ll} q^0_{PA} & \overset{\pi^k_{PA}}{\rightsquigarrow} \quad q^k_{PA} \\ I_{PA} & \overset{\pi^k_{PA}}{\rightsquigarrow} \quad q^k_{PA} \text{ (since } q^k_{PA\,a} \in I_{PA}\text{)} \end{array}$$

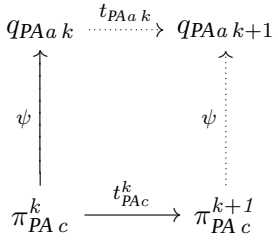
Lemma 6.1.1 describes path abstraction and states its correctness.

6.1.1 Lemma (Correctness Of Path Abstraction)**.** If the abstraction function $\psi : \Pi(pa_c) \to Q_{PA\,a}$ satisfies the following properties:

- ψ preserves the label of the last state on the path:

$$\psi(\pi^k_{PA\,c}) = q_{PA\,a\,k} \Longrightarrow q^k_{PA\,c} =_{PA} q_{PA\,a\,k} \tag{6.5}$$

- ψ makes the diagram commute:



(6.6)

- ψ preserves initial states:

$$q^0_{PAc} \in I_{PAc} \Longrightarrow \psi(q^0_{PAc}) \in I_{PAa} \tag{6.7}$$

- ψ has the additional property, related to simulation on parcel automata:

$$\left(\begin{array}{rcll} \pi^k_{PA\,c\,1} & = & q^0_{PAc} \xrightarrow{t^0_{PAc}} \cdots \xrightarrow{t^{k-1}_{PAc}} q^k_{PAc\,1} & \wedge \\ \pi^k_{PA\,c\,2} & = & q^0_{PAc} \xrightarrow{t^0_{PAc}} \cdots \xrightarrow{t^{k-1}_{PAc}} q^k_{PAc\,2} & \wedge \\ q^k_{PAc\,1} & \subseteq & q^k_{PAc\,2} & \end{array}\right) \Longrightarrow \psi(\pi^k_{PA\,c\,1}) \subseteq \psi(\pi^k_{PA\,c\,2}) \tag{6.8}$$

then

$$\mathcal{L}(pa_c) \quad \subseteq_{PA} \quad \mathcal{L}(pa_a) \tag{6.9}$$

$$pa_c \quad \preceq_{PA} \quad pa_a \tag{6.10}$$

Path abstraction is driven by exploration of concrete paths in the parcel automaton. Therefore the abstraction function is defined on the set of concrete paths onto abstract states: $\psi : \Pi(pa_c) \rightarrow Q_{PAa}$.

In Figure 6.1 we recall the parcel automaton of the $AndOr$ example that we first presented in Section 4.1. We use this example to illustrate path abstraction. Figure 6.2 describes an abstract automaton that satisfies the conditions of Lemma 6.1.1. The abstract automaton has a tree like structure — with the exception of the transitions leading to its final state, corresponding to the depth-first-search tree of paths through the concrete parcel automaton.

Table 6.1 describes the mapping ψ that maps the paths through the concrete parcel automaton to abstract states. For instance, line 13 in the table describes the path π^3 that is composed of the path π^2

$$\emptyset \longrightarrow r_1 = 00 \xrightarrow{c = 01} r_2 = 0000$$

followed by the transition

$$\xrightarrow{v = 0}$$

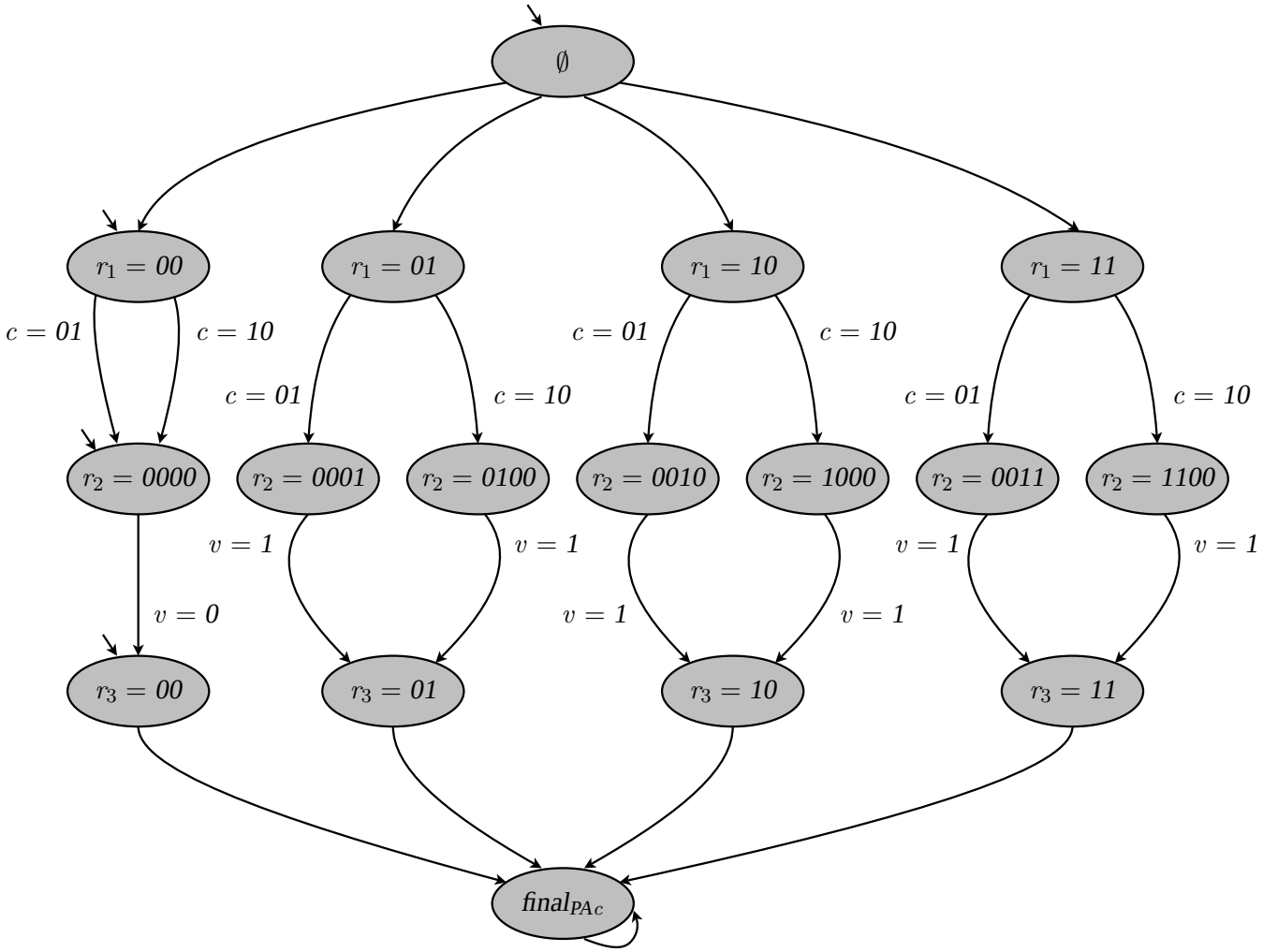


Figure 6.1: $AndOr$ Parcel Automaton.

leading to the state

$$r_3 = 00$$

The path π^2 in Equation 6.1 is described at line 5 in the table. The mapping of the path π^3 to an abstract state is $\psi(\pi^3) = \{\, r_3 = \alpha_2 \,\}$. Corresponding to Equation 6.11, we have the commuting

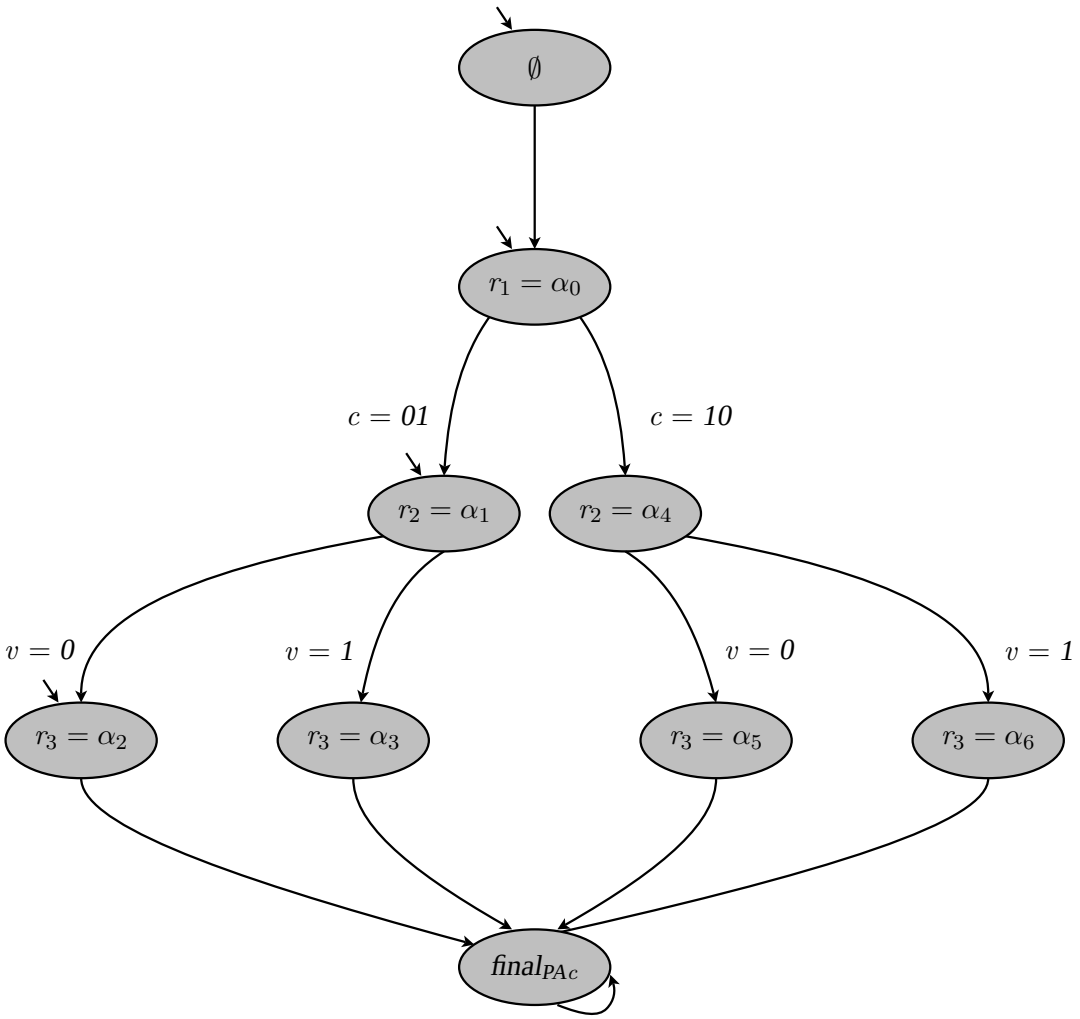


Figure 6.2: An abstract $AndOr$ parcel automaton.

diagram:

$$\begin{array}{ccc} r_1 = \alpha_1 & \xrightarrow{v=0} & r_1 = \alpha_2 \\ \uparrow \psi & & \uparrow \psi \\ \pi^2 & \xrightarrow{v=0} & \pi^3 \end{array} \tag{6.11}$$

Proof. We prove Equation 6.10 holds. This implies Equation 6.9 also holds. For simplicity we assume all states in Q_{PAc} are reachable from an initial state. Non-reachable states do not affect the behaviour of the parcel automaton so we can safely ignore them.

#	Concrete Path	Transition	Next State	Abstract State	Predecessor
0	$\emptyset$			$\emptyset$	
1	$\emptyset$	$\longrightarrow$	$r_1 = \mathit{00}$	$r_1 = \alpha_0$	0
2	$\emptyset$	$\longrightarrow$	$r_1 = \mathit{01}$	$r_1 = \alpha_0$	0
3	$\emptyset$	$\longrightarrow$	$r_1 = \mathit{10}$	$r_1 = \alpha_0$	0
4	$\emptyset$	$\longrightarrow$	$r_1 = \mathit{11}$	$r_1 = \alpha_0$	0
5	$\emptyset \longrightarrow r_1 = \mathit{00}$	$\overset{c=01}{\longrightarrow}$	$r_2 = \mathit{0000}$	$r_2 = \alpha_1$	1
6	$\emptyset \longrightarrow r_1 = \mathit{00}$	$\overset{c=10}{\longrightarrow}$	$r_2 = \mathit{0000}$	$r_2 = \alpha_4$	1
7	$\emptyset \longrightarrow r_1 = \mathit{01}$	$\overset{c=01}{\longrightarrow}$	$r_2 = \mathit{0001}$	$r_2 = \alpha_1$	2
8	$\emptyset \longrightarrow r_1 = \mathit{01}$	$\overset{c=10}{\longrightarrow}$	$r_2 = \mathit{0100}$	$r_2 = \alpha_4$	2
9	$\emptyset \longrightarrow r_1 = \mathit{10}$	$\overset{c=01}{\longrightarrow}$	$r_2 = \mathit{0010}$	$r_2 = \alpha_1$	3
10	$\emptyset \longrightarrow r_1 = \mathit{10}$	$\overset{c=10}{\longrightarrow}$	$r_2 = \mathit{1000}$	$r_2 = \alpha_4$	3
11	$\emptyset \longrightarrow r_1 = \mathit{11}$	$\overset{c=01}{\longrightarrow}$	$r_2 = \mathit{0011}$	$r_2 = \alpha_1$	4
12	$\emptyset \longrightarrow r_1 = \mathit{11}$	$\overset{c=10}{\longrightarrow}$	$r_2 = \mathit{1100}$	$r_2 = \alpha_4$	4
13	$\emptyset \longrightarrow r_1 = \mathit{00} \overset{c=01}{\longrightarrow} r_2 = \mathit{0000}$	$\overset{v=0}{\longrightarrow}$	$r_3 = \mathit{00}$	$r_3 = \alpha_2$	5
14	$\emptyset \longrightarrow r_1 = \mathit{00} \overset{c=10}{\longrightarrow} r_2 = \mathit{0000}$	$\overset{v=0}{\longrightarrow}$	$r_3 = \mathit{00}$	$r_3 = \alpha_5$	6
15	$\emptyset \longrightarrow r_1 = \mathit{01} \overset{c=01}{\longrightarrow} r_2 = \mathit{0001}$	$\overset{v=1}{\longrightarrow}$	$r_3 = \mathit{01}$	$r_3 = \alpha_3$	7
16	$\emptyset \longrightarrow r_1 = \mathit{01} \overset{c=10}{\longrightarrow} r_2 = \mathit{0100}$	$\overset{v=1}{\longrightarrow}$	$r_3 = \mathit{01}$	$r_3 = \alpha_6$	8
17	$\emptyset \longrightarrow r_1 = \mathit{10} \overset{c=01}{\longrightarrow} r_2 = \mathit{0010}$	$\overset{v=1}{\longrightarrow}$	$r_3 = \mathit{10}$	$r_3 = \alpha_3$	9
18	$\emptyset \longrightarrow r_1 = \mathit{10} \overset{c=10}{\longrightarrow} r_2 = \mathit{1000}$	$\overset{v=1}{\longrightarrow}$	$r_3 = \mathit{10}$	$r_3 = \alpha_6$	10
19	$\emptyset \longrightarrow r_1 = \mathit{11} \overset{c=01}{\longrightarrow} r_2 = \mathit{0011}$	$\overset{v=1}{\longrightarrow}$	$r_3 = \mathit{11}$	$r_3 = \alpha_3$	11
20	$\emptyset \longrightarrow r_1 = \mathit{11} \overset{c=10}{\longrightarrow} r_2 = \mathit{1100}$	$\overset{v=1}{\longrightarrow}$	$r_3 = \mathit{11}$	$r_3 = \alpha_6$	12
21	$\emptyset \longrightarrow r_1 = \mathit{00} \overset{c=01}{\longrightarrow} r_2 = \mathit{0000} \overset{v=0}{\longrightarrow} r_3 = \mathit{00}$	$\longrightarrow$	final_{PAc}	$r_1 = \mathit{final}_{PAa}$	13
23	$\emptyset \longrightarrow r_1 = \mathit{00} \overset{c=10}{\longrightarrow} r_2 = \mathit{0000} \overset{v=0}{\longrightarrow} r_3 = \mathit{00}$	$\longrightarrow$	final_{PAc}	$r_1 = \mathit{final}_{PAa}$	14
23	$\emptyset \longrightarrow r_1 = \mathit{01} \overset{c=01}{\longrightarrow} r_2 = \mathit{0001} \overset{v=1}{\longrightarrow} r_3 = \mathit{01}$	$\longrightarrow$	final_{PAc}	$r_1 = \mathit{final}_{PAa}$	15
24	$\emptyset \longrightarrow r_1 = \mathit{01} \overset{c=10}{\longrightarrow} r_2 = \mathit{0100} \overset{v=1}{\longrightarrow} r_3 = \mathit{01}$	$\longrightarrow$	final_{PAc}	$r_1 = \mathit{final}_{PAa}$	16
25	$\emptyset \longrightarrow r_1 = \mathit{10} \overset{c=01}{\longrightarrow} r_2 = \mathit{0010} \overset{v=1}{\longrightarrow} r_3 = \mathit{10}$	$\longrightarrow$	final_{PAc}	$r_1 = \mathit{final}_{PAa}$	17
26	$\emptyset \longrightarrow r_1 = \mathit{10} \overset{c=10}{\longrightarrow} r_2 = \mathit{1000} \overset{v=1}{\longrightarrow} r_3 = \mathit{10}$	$\longrightarrow$	final_{PAc}	$r_1 = \mathit{final}_{PAa}$	18
27	$\emptyset \longrightarrow r_1 = \mathit{11} \overset{c=01}{\longrightarrow} r_2 = \mathit{0011} \overset{v=1}{\longrightarrow} r_3 = \mathit{11}$	$\longrightarrow$	final_{PAc}	$r_1 = \mathit{final}_{PAa}$	19
28	$\emptyset \longrightarrow r_1 = \mathit{11} \overset{c=10}{\longrightarrow} r_2 = \mathit{1100} \overset{v=1}{\longrightarrow} r_3 = \mathit{11}$	$\longrightarrow$	final_{PAc}	$r_1 = \mathit{final}_{PAa}$	20

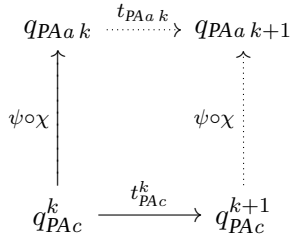
Table 6.1: The path map ψ.

We consider the relation $\chi \subseteq Q_{PAc} \times \Pi(pa_c)$ defined by

$$\chi \equiv \{\, (q^k_{PAc}, \pi^k_{PA\,c}) \mid \pi^k_{PA\,c} \text{ ends with } q^k_{PAc} \,\}$$

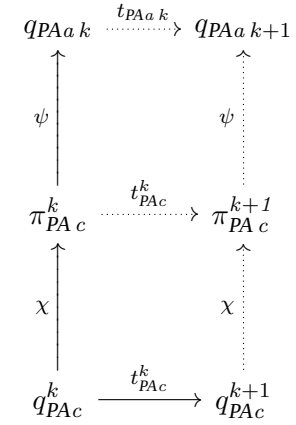
We show that $\psi \circ \chi$ is a parcel automata simulation relation.

Commuting Diagram We show the following diagram commutes:



(6.12)

The diagram in Equation 6.12 is obtained by combining the two diagrams in Equation 6.13. The lower part commutes according to the definition of χ and the upper part according to Equation 6.11.



(6.13)

Additional Property The relation $\psi \circ \chi$ must satisfy Equation 4.9 in the definition of simulation on parcel automata. For this purpose we use Equation 6.8 satisfied by ψ. Consider

$$(q^k_{PAc\,2}, q_{PAa\,k\,2}) \in \psi \circ \chi \tag{6.14}$$

$$q^k_{PAc\,1} \subseteq q^k_{PAc\,2} \tag{6.15}$$

We must show there exists $q_{PAa\,k\,1}$ so that:

$$q_{PAa\,k\,1} \subseteq q_{PAa\,k\,2} \tag{6.16}$$

$$(q^k_{PAc\,1}, q_{PAa\,k\,1}) \in \psi \circ \chi \tag{6.17}$$

From Equation 6.14 we infer there must exist $\pi^k_{PA\,c\,2} \in \Pi(pa_c)$ such that:

$$\begin{aligned} (q^k_{PAc\,2}, \pi^k_{PA\,c\,2}) &\in \chi \\ \pi^k_{PA\,c\,2} &= q^0_{PAc} \xrightarrow{t^0_{PAc}} \cdots \xrightarrow{t^{k-1}_{PAc}} q^k_{PAc\,2} \\ \psi(\pi^k_{PA\,c\,2}) &= q_{PAa\,k\,2} \end{aligned} \tag{6.18}$$

Equation 6.18 implies that there exists $\pi^k_{PA\,c\,1} \in \Pi(pa_c)$ such that:

$$\begin{aligned}(q^k_{PAc\,1}, \pi^k_{PA\,c\,1}) &\in \chi \\ \pi^k_{PA\,c\,1} &= q^0_{PAc} \xrightarrow{t^0_{PAc}} \cdots \xrightarrow{t^{k-1}_{PAc}} q^k_{PAc\,1}\end{aligned}$$

Applying Equation 6.8 to $\pi^k_{PA\,c\,1}$ and $\pi^k_{PA\,c\,2}$ we have that

$$\psi(\pi^k_{PA\,c\,1}) \subseteq \psi(\pi^k_{PA\,c\,2})$$

If we set

$$q_{PAa\,k\,1} = \psi(\pi^k_{PA\,c\,1})$$

then both Equation 6.16 and Equation 6.17 hold.

Initial States Let $q^0_{PAc} \in I_{PAc}$. We must show that there exists $q_{PAa\,0} \in I_{PAa}$ so that

$$(q^0_{PAc}, q_{PAa\,0}) \in \psi \circ \chi$$

Since $\pi^0_{PA\,c} = q^0_{PAc}$ is a path to q^0_{PAc} we have

$$(q^0_{PAc}, q^0_{PAc}) \in \chi$$

Applying Equation 6.7 to q^0_{PAc} we have:

$$\psi(q^0_{PAc}) \in I_{PAa}$$

For $q_{PAa\,0} = \psi(q^0_{PAc})$ we get the desired conclusion.

□

We say two parcel transitions are equivalent if they have equivalent transition labels.

$$(q_{PA1}, t_{PA1}, q_{PA1}{}') =_{PA} (q_{PA2}, t_{PA2}, q_{PA2}{}') \equiv (t_{PA1} =_{PA} t_{PA2})$$

We define path equivalence between two paths if they both start from equivalent states, have the

same length and have pairwise equivalent transitions:

$$
\begin{array}{l}
\pi^{k}_{PA\,1} =_{PA} \pi^{l}_{PA\,2} \\
\quad \equiv \\
\qquad (k = l) \wedge (\mathit{dom}\, q^{0}_{PA\,1} = \mathit{dom}\, q^{0}_{PA\,2}) \wedge \\
\qquad\quad \Big(\; k > 0 \Longrightarrow \forall\, j \in \{\, 0,\, \ldots,\, k-1 \,\}.\; (q^{j}_{PA\,1}, t^{j}_{PA\,1}, q^{j+1}_{PA\,1}) =_{PA} (q^{j}_{PA\,2}, t^{j}_{PA\,2}, q^{j+1}_{PA\,2}) \; \Big)
\end{array}
$$

Path equivalence is related to run equivalence. However, because we use path equivalence in the context of closed parcel automata we do not require in its definition that the states at the same index be equivalent.

Parcel computations are infinite. Termination of the abstraction algorithm we present in Section 6.5 depends on whether the datapath computations that affect the parcel's state are confined to a finite prefix of the parcel run . From a point on in the run, a parcel's state is updated only by copying of values from the previous state.

$StateFanOut^{k}$ denotes the set of variables in the fan-out of the current state registers at step $(q^{k}_{PA}, t^{k}_{PA}, q^{k+1}_{PA})$:

Base Case

$$\mathit{dom}\, q^{k}_{PA} \subseteq StateFanOut^{k}$$

Inductive Case

$$\forall\, (v_1, b, v_2) \in \mathit{fg}^{k}.\mathit{Succ}.\; v_1 \in StateFanOut^{k} \wedge v_2 \notin \mathit{PclN} \Longrightarrow v_2 \in StateFanOut^{k}$$

6.1.2 Definition (Terminating Run)**.** The run σ_{PA} is terminating if it either contains the final state final_{PA} or there exists $n \geq 0$ such that for all $k \geq n$, in the step $(q^{k}_{PA}, t^{k}_{PA}, q^{k+1}_{PA})$ only current state values are copied into next-state parcel registers.

$$\forall\, k \geq n.\; \mathit{dom}\, q^{k+1}_{PA} \left[{}^{\mathit{NextRegPcl}}/_{\mathit{RegPcl}} \right] \subseteq StateFanOut^{k} \tag{6.19}$$

We can think of Equation 6.19 to cover termination when the parcel exits the pipeline and its state becomes empty since we can define the domain of the final state to be the empty set. If the run σ_{PA} contains the final state then there exists $n \geq 1$ so that

$$
\begin{array}{rcl}
\forall\, k \geq n.\; q^{k}_{PA} & = & \mathit{final}_{PA} \\
\forall\, k \geq n.\; t^{k}_{PA} & = & \emptyset \\
\forall\, k \geq n.\; \mathit{dom}\, \mathit{final}_{PA} \left[{}^{\mathit{NextRegPcl}}/_{\mathit{RegPcl}} \right] & \subseteq & \emptyset
\end{array}
$$

Equation 6.19 states that datapath computed values, constants, **choice** and inputs do not propagate into the parcel's next state. For instance, a computation in which a parcel stalls indefinitely is terminating. As a generalization, the definition could be relaxed to allow propagation of constants.

The notion of the driver of a variable at step k is used to analyze terminating runs. We define $driver\ (v_2, k)$ to be (v_1, n) such that variable v_1 at step n propagates through copying into the value of v_2 at step k. The definition is inductive.

6.1.3 Definition (Driver)**. Base Case** $k = 0 \wedge v \in \mathit{dom}\, q^0_{PA}$

$$\forall\, v \in \mathit{dom}\, q^0_{PA}.\ driver\ (v, 0) = (v, 0)$$

Inductive Case $k \geq 0$

$$\begin{array}{rcl}
\forall\, v \in (\mathit{roots}\, fg^k) \setminus \mathit{RegPcl}.\ driver\ (v, k) & = & (v, k) \\
\forall\, (v_1, b, v_2) \in fg^k.\mathit{Succ}.\ v_2 \notin \mathit{PclN} \Longrightarrow driver\ (v_2, k) & = & driver\ (v_1, k) \\
\forall\, (w, b, v) \in fg^k.\mathit{Succ}.\ driver\ (v, k) & = & driver\ (v, k) \\
\forall\, v \in \mathit{PclN}.\ driver\ (v, k) & = & (v, k) \\
\forall\, v \in \mathit{dom}\, q^k_{PA}.\ driver\ (v, k+1) & = & driver\ (v', k)
\end{array}$$

We define the equivalence '$=_{\mathit{driver}}$' on the parcel automaton states of the run σ_{PA}.

$$\begin{array}{l}
q^{k_1}_{PA} =_{\mathit{driver}} q^{k_2}_{PA} \equiv \\
\quad (q^{k_1}_{PA} =_{PA} q^{k_2}_{PA}) \wedge \forall\, v \in \mathit{dom}\, q^{k_1}_{PA}.\ driver\ (v, k_1) = driver\ (v, k_2)
\end{array}$$

The equivalence is stronger than state equality:

$$q^{k_1}_{PA} =_{\mathit{driver}} q^{k_2}_{PA} \Longrightarrow q^{k_1}_{PA} = q^{k_2}_{PA}$$

We extend '$=_{\mathit{driver}}$' to transitions:

$$\begin{array}{l}
(q^{k_1}_{PAa}, t^{k_1}_{PAa}, q^{k_1+1}_{PAa}) =_{\mathit{driver}} (q^{k_2}_{PAa}, t^{k_2}_{PAa}, q^{k_2+1}_{PAa}) \equiv \\
\quad (q^{k_1}_{PA} =_{\mathit{driver}} q^{k_2}_{PA}) \wedge (t^{k_1}_{PAa} =_{PA} t^{k_2}_{PAa}) \wedge (q^{k_1+1}_{PA} =_{PA} q^{k_2+2}_{PA})
\end{array}$$

Since $t^{k_1}_{PAa}$ and $t^{k_2}_{PAa}$ are equivalent, they specify the same copying of values into next states so it follows that

$$(q^{k_1}_{PAa}, t^{k_1}_{PAa}, q^{k_1+1}_{PAa}) =_{\mathit{driver}} (q^{k_2}_{PAa}, t^{k_2}_{PAa}, q^{k_2+1}_{PAa}) \Longrightarrow q^{k_1+1}_{PA} =_{\mathit{driver}} q^{k_2+1}_{PA}$$

Before we present the next lemma, we recall the concept of closed parcel automaton (Definition 4.6.2). The closure of a parcel automaton does not influence the datapath behaviours it specifies. Our proofs rely in some cases on the parcel automata to be closed. Since the end result of obtaining an abstract parcel automaton is to perform abstract interpretation of the datapath, the closure of the parcel automaton is not implemented. Instead, the closure performed only in proofs.

Lemma 6.1.4 shows that under sufficient conditions, path abstraction preserves language equality. The essential condition is that for each path in the abstract parcel automaton there exists an equivalent one in the concrete automaton.

6.1.4 Lemma. If the following conditions hold:

- The parcel automaton pa_c is closed.
- All runs of pa_c are terminating.
- For every prefix in $\Pi(pa_a)$ there exists an equivalent one in $\Pi(pa_c)$:

$$\forall\, \pi^k_{PA\,a} \in \Pi(pa_a).\ \exists\, \pi^k_{PA\,c} \in \Pi(pa_c).\ \pi^k_{PA\,c} =_{PA} \pi^k_{PA\,a} \tag{6.20}$$

then

$$\mathcal{L}(pa_a) \subseteq_{PA} \mathcal{L}(pa_c) \tag{6.21}$$

Proof. Consider $\sigma_{PAa} \in \mathcal{L}(pa_a)$. We need to show there exists $\sigma_{PAc} \in \mathcal{L}(pa_c)$ so that

$$\sigma_{PAa} =_{PA} \sigma_{PAc} \tag{6.22}$$

Case 1 σ_{PAa} is non-terminating. We prove this leads to a contradiction.

If σ_{PAa} is non-terminating it has increasingly longer prefixes $\pi^{k_i}_{PA\,a}$ of form

$$\pi^{k_i}_{PA\,a} = q^0_{PAa} \xrightarrow{t^0_{PAa}} \cdots \xrightarrow{t^{k_i-1}_{PAa}} q^{k_i}_{PAa}$$

such that the number of parcel transitions that contain datapath computations is at least i.

Applying Equation 6.20 to each prefix $\pi^{k_i}_{PA\,a}$ we obtain an equivalent prefix in $\Pi(pa_c)$:

$$\begin{array}{lll} \pi^{k_i}_{PA\,c} & = & q^0_{PAc} \xrightarrow{t^0_{PAc}} \cdots \xrightarrow{t^{k_i-1}_{PAc}} q^{k_i}_{PAc} \\ \pi^{k_i}_{PA\,a} & =_{PA} & \pi^{k_i}_{PA\,c} \end{array} \tag{6.23}$$

Equation 6.28 implies that the path $\pi^{k_i}_{PA\,c}$ also has at least i transitions that contain datapath computations. For $i > |Q_{PAc}|$ there exists one state q^j_{PAc} that performs two different transitions both of which contain datapath computations. The path $\pi^{k_i}_{PA\,c}$ expresses as

$$\pi^{k_i}_{PA\,c} = q^0_{PAc} \xrightarrow{t^0_{PAc}} \cdots \xrightarrow{t^{j-1}_{PAc}} q^j_{PAc} \xrightarrow{t^j_{PAc}} \cdots\, q^{j+l}_{PAc} \xrightarrow{t^{j+l}_{PAc}} \cdots \text{ with } q^j_{PAc} = q^{j+l}_{PAc}$$

The following is a non-terminating run of pa_c:

$$q^0_{PAc} \xrightarrow{t^0_{PAc}} \cdots \xrightarrow{t^{j-1}_{PAc}} q^j_{PAc} \xrightarrow{t^j_{PAc}} \cdots\, q^j_{PAc} \xrightarrow{t^j_{PAc}} \cdots$$

Case 2 σ_{PAa} is terminating.

If it contains the state final_{PAa}, then by selecting a prefix $\pi^n_{PA\,a}$ so that $q^n_{PAa} = \mathit{final}_{PAa}$ and applying Equation 6.20 we obtain $\pi^n_{PA\,c}$ so that $\pi^n_{PA\,c} =_{PA} \pi^n_{PA\,a}$. This concrete path is trivially extended to a concrete run matching σ_{PAa}.

Consider now the case when final_{PAa} does not appear in the run σ_{PAa}. The run then consists of the finite prefix $\pi^n_{PA\,a}$ followed by an infinite suffix of transitions that only update the parcel's state by copying.

$$\begin{aligned} \sigma_{PAa} &= q^0_{PAa} \xrightarrow{t^0_{PAa}} \cdots \xrightarrow{t^{n-1}_{PAa}} q^n_{PAa} \xrightarrow{t^n_{PAa}} q^{n+1}_{PAa} \cdots \\ \pi^n_{PA\,a} &= q^0_{PAa} \xrightarrow{t^0_{PAa}} \cdots \xrightarrow{t^{n-1}_{PAa}} q^n_{PAa} \end{aligned}$$

Since the R_{PAa} is finite, in the infinite suffix

$$q^n_{PAa} \xrightarrow{t^n_{PAa}} q^{n+1}_{PAa} \xrightarrow{t^{n+1}_{PAa}} q^{n+2}_{PAa} \cdots$$

we have

$$\forall\, k_2 \geq n.\ \forall\, v_2 \in \mathit{dom}\, q^k_{PAa}.\ \exists\, k_1 \leq n.\ \exists\, v_1.\ \mathit{driver}\,(v_2, k_2) = (v_1, k_1)$$

Since the indices k_1 are bounded by n, there exists an index $s > n$ such that from then on, all transitions are '$=_{\mathit{driver}}$' equivalent to transitions at an index between n and $s-1$:

$$\begin{aligned} &\exists\, s > n. \\ &\quad \forall\, k \geq s.\ \exists\, l \in \{\, n, \ldots, s-1 \,\}.\ (q^k_{PAa}, t^k_{PAa}, q^{k+1}_{PAa}) =_{\mathit{driver}} (q^l_{PAa}, t^l_{PAa}, q^{l+1}_{PAa}) \end{aligned} \tag{6.24}$$

Applying Equation 6.20 to the prefix $\pi^s_{PA\,a}$ we obtain $\pi^s_{PA\,c}$ such that:

$$\begin{array}{lll} \pi^s_{PA\,c} & =_{PA} & \pi^s_{PA\,a} \\ \pi^s_{PA\,c} & = & q^0_{PAc} \xrightarrow{t^0_{PAc}} \cdots \xrightarrow{t^{s-1}_{PAc}} q^s_{PAc} \end{array}$$

Since pa_c is closed we can assume that the path $\pi^s_{PA\,c}$ has the property that

$$\forall\, k \in \{\, 0, \ldots, s \,\}.\; q^k_{PAc} =_{PA} q^k_{PAa}$$

We prove by induction that for $k \geq n$, $\pi^k_{PA\,c}$ can be extended to a path $\pi^{k+1}_{PA\,c}$ that is equivalent to the prefix path of length $k+1$ of σ_{PAa}.

First, we prove a helping result:

$$\begin{array}{l} \forall\, k \geq s. \\ \quad \pi^k_{PA\,c} =_{PA} \pi^k_{PA\,a} \\ \quad\quad \Longrightarrow \\ \quad\quad\quad \forall\, l_1 \in \{\, n, \ldots, k \,\}.\, \forall\, l_2 \in \{\, n, \ldots, k \,\}. \\ \quad\quad\quad\quad (q^{l_1}_{PAa} =_{driver} q^{l_2}_{PAa}) \Longrightarrow q^{l_1}_{PAc} = q^{l_2}_{PAc} \end{array} \tag{6.25}$$

Since $\pi^k_{PA\,c} =_{PA} \pi^k_{PA\,a}$ it means that for any $l \in \{\, n, \ldots, k \,\}$ the transitions from q^n_{PAa} to q^l_{PAa} have the same copying effect as the transitions from q^n_{PAc} to q^l_{PAc}. Since '$=_{driver}$' captures the make up of states $q^{l_1}_{PAa}$ and $q^{l_2}_{PAa}$ in terms of the values in state q^n_{PAa}, it follows that if $q^{l_1}_{PAa} =_{driver} q^{l_2}_{PAa}$ then $q^{l_1}_{PAc} = q^{l_2}_{PAc}$.

Base Case $k = n$. Since $\pi^s_{PA\,c} =_{PA} \pi^s_{PA\,a}$ and $n < s$, $\pi^k_{PA\,c}$ is a prefix of $\pi^s_{PA\,c}$ and it therefore extends so that $\pi^{k+1}_{PA\,c} =_{PA} \pi^{k+1}_{PA\,a}$.

Inductive Case

If $k < s$ then $\pi^k_{PA\,c}$ is still a strict prefix of $\pi^s_{PA\,c}$ and therefore the extension is done as in the base case.

If $k \geq s$ then there exists $l \in \{\, n, \ldots, s-1 \,\}$ so that

$$(q^k_{PAa}, t^k_{PAa}, q^{k+1}_{PAa}) =_{driver} (q^l_{PAa}, t^l_{PAa}, q^{l+1}_{PAa})$$

Using the inductive hypothesis we have:

$$(q^l_{PAa}, t^l_{PAa}, q^{l+1}_{PAa}) \quad =_{PA} \quad (q^l_{PAc}, t^l_{PAc}, q^{l+1}_{PAc}) \text{ (since } l < k)$$

$$q^l_{PA\,c} \quad = \quad q^k_{PA\,c} \text{ (since } q^l_{PA\,a} =_{driver} q^k_{PA\,a})$$

We can therefore extend the path $\pi^k_{PA\,c}$ with the step $(q^l_{PA\,c}, t^l_{PA\,c}, q^{l+1}_{PA\,c})$:

$$\pi^{k+1}_{PA\,c} \equiv \pi^k_{PA\,c} \xrightarrow{t^l_{PA\,c}} q^{l+1}_{PA\,c}$$

We use the set of paths $\{\, \pi^i_{PA\,c} \mid i \geq 0 \,\}$ to define $\sigma_{PA\,c}$ so that $\sigma_{PA\,c} =_{PA} \sigma_{PA\,a}$.

□

6.2 Approximating The Induced Parcel Automaton

The parcel automaton induced by the parcel map contains all parcel steps that occur in pipeline computations and it describes exactly the datapath computations that arise during pipeline computations. Since the steps of the induced parcel automaton are defined by the steps of the pipeline, its definition needs the inductive run of the entire pipeline which is not practical. We therefore approximate the induced parcel automaton with another one that is defined inductively using approximations of the parcel steps of the induced parcel automaton. This definition is simple and efficient and can be used in the abstraction algorithm.

The approximate parcel automaton is conservative since its set of states and transitions include the ones of the induced parcel automaton. Given two parcel automata

$$\begin{aligned} pa_1 &= \langle Q_{PA1}, R_{PA1}, T_{PA1}, I_{PA1} \rangle \\ pa_2 &= \langle Q_{PA2}, R_{PA2}, T_{PA2}, I_{PA2} \rangle \end{aligned}$$

We define

$$pa_1 \subseteq pa_2 \equiv (R_{PA1} \subseteq R_{PA2}) \wedge (I_{PA1} \subseteq I_{PA2})$$

6.2.1 Proposition. If $pa_1 \subseteq pa_2$ then

$$\begin{array}{rcl} pa_1 & \preceq_{PA} & pa_2 \\ \mathcal{L}(pa_1) & \subseteq_{PA} & \mathcal{L}(pa_2) \end{array}$$

A parcel automaton $pa \in Pa(Pipe)$ *covers* the datapath computations of the pipeline model $Pipe$ if $pa(Pipe, PclMap) \subseteq pa$. A parcel automaton that covers the datapath computations is a conservative approximation according to Proposition 6.2.1. Our approximation is based on approximations

of the parcel map and of the parcel steps that occur in the pipeline model computations.

In the remainder of the section we describe a technique to define the approximate parcel automaton $pa_{c\,1}$ that covers the datapath computations:

$$\mathit{pa}(Pipe_c, PclMap) \subseteq pa_{c\,1}$$

A parcel step is determined by the parcel's representation as a set of variables, the parcel's current state, the fan-out graph and the parcel and control environments. A parcel step is executed within a pipeline step. The control variables that determine the parcel's execution are constrained by control assignments and datapath computed control values. In the definition of the states and steps of the automaton $pa_{c\,1}$ we substitute, for the control environment contained in the step of the pipeline model, an environment that is easier to compute. We denote it in the following equations by $e_{ctrl\ step} \in \mathit{PEnv}(V_{ctrl} \cup \mathit{NextRegCtrl})$. The environment $e_{ctrl\ step}$ should respect the assignments to control variables:

$$e_{ctrl\ step} \models (Pipe_c.Tr) \mid_{\mathit{dom}\ e_{ctrl\ step}} \tag{6.26}$$

If parcel p has the parcel step $(q_{PAc}, \langle fg_c, e_{ctrlc}, e_{pclc}\rangle, q_{PAc}{}')$ in the pipeline model step $(q_{Pc}, t_{Pc}, q'_{Pc})$, then the fan-out graph fg_c is maximal. There are no fan-out edges (v_l, b, v_k) such that $v_l \in fg_c.\mathit{Nodes}$ and $(q_{Pc} \cup t_{Pc}) \models b$ but $v_k \notin fg_c.\mathit{Nodes}$. The fan-out graph fg_c is maximal under the environment $q_{Pc} \cup t_{Pc}$. The following predicate represents maximality of the fan-out graph on the transition label of the parcel step.

$$\begin{aligned} &\mathit{IsMaximalFanOut}\ (q_{PAc}, \langle fg_c, e_{ctrlc}, e_{pclc}\rangle, q_{PAc}{}')\ e_{ctrl\ step} \equiv \\ &\quad \forall\, (v_l, b, v_k) \in \mathit{FanOutEdges}. \\ &\qquad (v_l, b, v_k) \in fg_c.\mathit{Succ} \iff v_l \in fg_c.\mathit{Nodes} \wedge e_{ctrl\ step} \models b \end{aligned} \tag{6.27}$$

The control environment $e_{ctrl\ step}$ that determines the fan-out graph of the parcel and influences its datapath computations may contain register variables that in a pipeline computation are influenced by the previous step of the parcel. If such variables are not constrained in the definition of $e_{ctrl\ step}$, then the automaton $pa_{c\,1}$ may have parcel steps and computations that are not reachable in the induced (precise) parcel automaton. These non-reachable computations do not affect the datapath circuits that $pa_{c\,1}$ specifies, however they do affect an abstraction algorithm: first, by enlarging the search space and second, by affecting termination, if $pa_{c\,1}$ has non-terminating runs that are not possible in the induced automaton.

Figure 6.3 describes two different pipeline models that both have unreachable datapath computations. In both cases a parcel produces in the current parcel step a control value that later influences its datapath computation in the next parcel step. For the pipeline model in Figure 6.3a, the con-

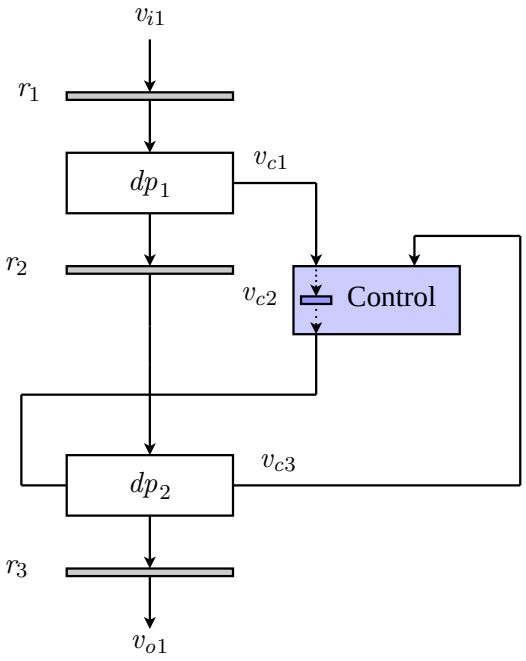


Fig. 6.3a.

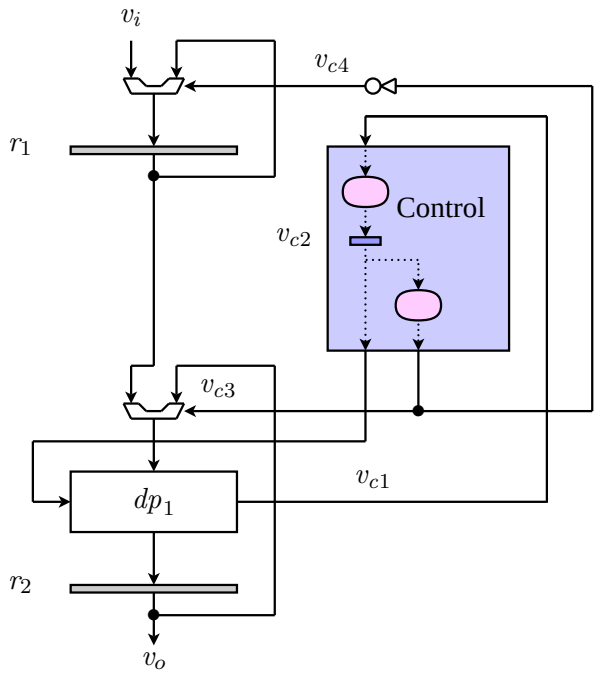


Fig. 6.3b.

Figure 6.3: Pipeline models with unreachable datapath computations.

trol output v_{c1} produced by dp_1 and corresponding to a parcel $p = \{ r_1 \}$ is saved into the control register v_{c2}. At the next step, the value of v_{c2} influences the parcel step of parcel $p = \{ r_2 \}$ as it propagates through dp_2. Therefore, in a parcel computation, the output v_{c1} of dp_1 and the input v_{c2} of dp_2 are the same. The example in Figure 6.3b describes a pipeline model in which the datapath performs an iterative operation. Datapath dp_1 has both a control input and output. As the schematics of the pipeline control describe, the output of the datapath in the current step influences both the the control input in the next step and the fan-out graph, i.e. whether the parcel stalls. The variable v_{c2} represents the control state of the datapath computation. In a pipeline computation the variable converges to a final control state that ends the parcel computation.

The abstraction algorithm requires a set of parcels $Parcels \subseteq \mathcal{P}(\mathcal{P}(V_{pcl} \cup \mathit{NextRegPcl}))$ that approximates the parcel map:

$$\forall\, (q_{Pc}, t_{Pc}, q'_{Pc}) \in R_{Pc}.\ PclMap\ (q_{Pc}, t_{Pc}, q'_{Pc}) \in Parcels \tag{6.28}$$

We define the parcel step predicate $PclStep$ to be used in the definition of $pa_{c\,1}$. In the definition below we use the predicate $WellDefinedPclStep$ that stands for the definition of parcel steps (Definition 4.4.1). Consistency with respect to datapath behaviour (Equation 4.4) is represented by $ConsistentPclStep$. The parcel step predicate states whether the triplet $(q_{\mathit{PAc}}, \langle fg_c, e_{ctrlc}, e_{pclc} \rangle, {q_{\mathit{PAc}}}')$ is a parcel step that can execute under a control environment that satisfies control assignment constraints and maximality of fan-out graphs. The predicate also takes in consideration the propagation of the parcel's control state, that represents the control values generated by the parcel's step.

$$\begin{array}{l} PclStep\ (q_{\mathit{PAc}}, \langle fg_c, e_{ctrlc}, e_{pclc} \rangle, {q_{\mathit{PAc}}}')\ e_{ctrl\ step} \equiv \\ \left(\begin{array}{c} e_{ctrl\ step} \models (Pipe_c.\mathit{Tr}) \mid \mathit{dom}\ e_{ctrl\ step} \\ \wedge \\ e_{ctrlc} \subseteq e_{ctrl\ step} \end{array} \right) \\ \bigwedge \\ \left(\begin{array}{c} WellDefinedPclStep\ (q_{\mathit{PAc}}, \langle fg, e_{ctrl}, e_{pclc} \rangle, {q_{\mathit{PAc}}}') \\ \wedge \\ ConsistentPclStep\ (q_{\mathit{PAc}}, \langle fg, e_{ctrl}, e_{pclc} \rangle, {q_{\mathit{PAc}}}') \\ \wedge \\ \left(\begin{array}{c} \exists\, p \in Parcels.\ p \cap \mathit{RegPcl} \subseteq \mathit{dom}\ q_{\mathit{PAc}} \wedge \mathit{roots}\ fg = p \end{array} \right) \\ \wedge \\ IsMaximalFanOut\ (q_{\mathit{PAc}}, \langle fg_c, e_{ctrlc}, e_{pclc} \rangle, {q_{\mathit{PAc}}}')\ e_{ctrl\ step} \end{array} \right) \end{array} \tag{6.29}$$

The definition of the parcel automaton

$$pa_{c\,1} = \langle Q_{PAc\,1}, R_{PAc\,1}, T_{PAc\,1}, I_{PAc\,1} \rangle$$

is done by induction. The resulting parcel automaton depends on a function $ControlEnvDom$ that stands for the heuristical choice of the control variables that affect the parcel. When the definition of $PclStep$ is implemented in the abstraction algorithm, $ControlEnvDom$ will be chosen heuristically to trim the search space or to provide termination.

We prove that $pa_{c\,1}$ covers the datapath computations independently of the particular heuristic used. The set of reachable pairs of parcel and control states are denoted by $PclStatePairs$.

Base Case The set of initial states consists of all unconstrained parcel states.

$$I_{PAc\,1} = \{\, q_{PAc} \mid \exists\, p \in Parcels.\ \mathit{dom}\, q_{PAc} = p \cap \mathit{RegPcl} \,\} \tag{6.30}$$

$$PclStatePairs \subseteq I_{PAc\,1} \times \{\, \mathbf{true} \,\} \tag{6.31}$$

Inductive Case The set of states and transitions is defined inductively using the predicate $PclStep$.

$$\begin{array}{l} \forall\, q_{PAc}.\ \forall\, q_{ctrl}.\ \forall\, t_{PAc}.\ \forall\, {q_{PAc}}'. \\ \left(\begin{array}{l} (q_{PAc}, q_{ctrl}) \in PclStatePairs \wedge \\ \quad \exists\, e_{ctrl\,step} \in \mathit{PEnv}(V_{ctrl} \cup \mathit{NextRegCtrl}). \\ \qquad \mathit{dom}\, e_{ctrl\,step} = ControlEnvDom\ (q_{PAc}, q_{ctrl}, t_{PAc}) \wedge \\ \qquad\quad PclStep\ (q_{PAc}, t_{PAc}, {q_{PAc}}')\ e_{ctrl\,step} \wedge \\ \qquad\qquad \left(\ q_{ctrl} \cup {q_{ctrl}}' \left[\mathit{RegCtrl} / \mathit{NextRegCtrl} \right] \ \right) \subseteq e_{ctrl\,step} \end{array} \right) \\ \quad \Longrightarrow \\ \qquad ({q_{PAc}}', {q_{ctrl}}') \in PclStatePairs \wedge (q_{PAc}, t_{PAc}, {q_{PAc}}') \in R_{PAc\,1} \wedge {q_{PAc}}' \in Q_{PAc\,1} \end{array} \tag{6.32}$$

The domain of the control environment $e_{ctrl\,step}$ is chosen so that

$$\mathit{dom}\, q_{ctrl} \subseteq ControlEnvDom\ (q_{PAc}, q_{ctrl}, t_{PAc}) \tag{6.33}$$

As defined by Equation 6.32, the transition relation of $pa_{c\,1}$ contains only parcel steps. The closure $\overline{pa}_{c\,1}$ of $pa_{c\,1}$ includes the parcel transitions for such steps. We use the following notation for the closed automaton:

$$\overline{pa}_{c\,1} = (\overline{Q}_{PAc\,1}, \overline{R}_{PAc\,1}, \overline{T}_{PAc\,1}, \overline{I}_{PAc\,1})$$

Lemma 6.2.2 states that $\overline{pa}_{c\,1}$ covers the datapath computations.

6.2.2 Lemma.

$$\mathit{pa}(\mathit{Pipe}_c, \mathit{PclMap}) \subseteq \overline{\mathit{pa}}_{c\,1}$$

Proof. The claim is proved by induction. We show that for any pipeline step and any parcel within that step, there exists a control state so that the parcel and control state pair is in $\mathit{PclStatePairs}$.

$$\begin{array}{l}\forall\, q_{Pc} \in Q_{Pc}.\\ \quad \forall\, t_{Pc} \in T_{Pc}.\, \forall\, {q_{Pc}}' \in Q_{Pc}.\\ \qquad (q_{Pc}, t_{Pc}, q'_{Pc}) \in R_{Pc} \Longrightarrow \left(\begin{array}{l} \forall\, p \in \mathit{PclMap}\ (q_{Pc}, t_{Pc}, q'_{Pc}).\\ \quad \exists\, q_{PAc\,1} \in Q_{PAc\,1}.\, \exists\, q_{ctrl} \in \mathit{PEnv}(\mathit{RegCtrl}).\\ \qquad (q_{PAc\,1} \subseteq q_{Pc}) \wedge (q_{ctrl} \subseteq q_{Pc}) \wedge (\mathit{pclState}\ p \subseteq q_{PAc\,1})\\ \qquad\quad \wedge\, (q_{PAc\,1}, q_{ctrl}) \in \mathit{PclStatePairs}\end{array}\right)\end{array} \tag{6.34}$$

The other claim we prove by induction is that $R_{PAc} \subseteq \overline{R}_{PAc\,1}$.

$$\begin{array}{l}\forall\, q_{Pc} \in Q_{Pc}.\\ \quad \forall\, t_{Pc} \in T_{Pc}.\, \forall\, {q_{Pc}}' \in Q_{Pc}.\\ \qquad (q_{Pc}, t_{Pc}, q'_{Pc}) \in R_{Pc} \Longrightarrow \left(\begin{array}{l} \forall\, p \in \mathit{PclMap}\ (q_{Pc}, t_{Pc}, q'_{Pc}).\\ \quad (\mathit{pclState}\ p, \mathit{pclTrans}\ p, \mathit{pclNextState}\ p) \in \overline{R}_{PAc\,1}\end{array}\right)\end{array} \tag{6.35}$$

Base Case $q_{Pc} \in I_{Pc}$. From Equation 6.30 and Equation 6.31 we have

$$\begin{array}{rcl} I_{PAc} & \subseteq & I_{PAc\,1}\\ I_{PAc\,1} \times \{\,\mathbf{true}\,\} & \subseteq & \mathit{PclStatePairs}\end{array}$$

Since the parcel states of parcels within pipeline steps from initial pipeline states are initial, it follows that Equation 6.34 holds when $q_{Pc} \in I_{Pc}$.

Inductive Case Let $(q_{Pc}, t_{Pc}, q'_{Pc}) \in R_{Pc}$. By induction, Equation 6.36 holds of q_{Pc}.

$$\begin{array}{l} \forall\, p \in \mathit{PclMap}\ (q_{Pc}, t_{Pc}, q'_{Pc}).\\ \quad \exists\, q_{PAc\,1} \in Q_{PAc\,1}.\, \exists\, q_{ctrl} \in \mathit{PEnv}(\mathit{RegCtrl}).\\ \qquad (q_{PAc\,1} \subseteq q_{Pc}) \wedge (q_{ctrl} \subseteq q_{Pc}) \wedge (\mathit{pclState}\ p \subseteq q_{PAc\,1})\\ \qquad\quad \wedge\, (q_{PAc\,1}, q_{ctrl}) \in \mathit{PclStatePairs}\end{array} \tag{6.36}$$

We prove Equation 6.37 holds for the step $(q_{Pc}, t_{Pc}, {q'}_{Pc})$.

$$\begin{array}{l} \forall\, p \in PclMap\ (q_{Pc}, t_{Pc}, {q'}_{Pc}). \\ \quad (\mathit{pclState}\ p, \mathit{pclTrans}\ p, \mathit{pclNextState}\ p) \in \overline{R}_{PAc\,1} \end{array} \tag{6.37}$$

And that Equation 6.34 holds inductively for ${q_{Pc}}'$.

$$\forall\, {t_{Pc}}' \in T_{Pc}.\ \forall\, {q_{Pc}}'' \in Q_{Pc}.$$
$$({q'}_{Pc}, {t'}_{Pc}, {q''}_{Pc}) \in R_{Pc} \Longrightarrow \left(\begin{array}{l} \forall\, p' \in PclMap\ ({q'}_{Pc}, {t'}_{Pc}, {q''}_{Pc}). \\ \quad \exists\, {q_{PAc\,1}}' \in Q_{PAc\,1}.\ \exists\, {q_{ctrl}}' \in \mathit{PEnv}(\mathit{RegCtrl}). \\ \quad\quad ({q_{PAc\,1}}' \subseteq {q_{Pc}}') \wedge ({q_{ctrl}}' \subseteq {q_{Pc}}') \wedge (\mathit{pclState}\ p' \subseteq {q_{PAc\,1}}') \\ \quad\quad\quad \wedge ({q_{PAc\,1}}', {q_{ctrl}}') \in PclStatePairs \end{array} \right) \tag{6.38}$$

Part 1 We prove Equation 6.37.

Consider $p \in PclMap\ (q_{Pc}, t_{Pc}, {q'}_{Pc})$. Applying Equation 6.36, there exist $q_{PAc\,1}$ and q_{ctrl} so that

$$(q_{PAc\,1} \subseteq q_{Pc}) \wedge (q_{ctrl} \subseteq q_{Pc}) \wedge (\mathit{pclState}\ p \subseteq q_{PAc\,1}) \wedge (q_{PAc\,1}, q_{ctrl}) \in PclStatePairs \tag{6.39}$$

We let

$$\begin{array}{rcll} dom & = & ControlEnvDom\ q_{PAc\,1}\ q_{ctrl}\ (\mathit{pclTrans}\ p) & (6.40) \\ e_{ctrl\ step} & = & \left(\ q_{Pc} \cup t_{Pc} \cup {q_{Pc}}' \left[{V_r}' / V_r \right] \ \right) \mid_{dom} & (6.41) \\ {q_{ctrl}}' & \equiv & e_{ctrl\ step} \mid_{{V_r}'} & (6.42) \end{array}$$

and show that

$$PclStep\ (q_{PAc\,1}, \mathit{pclTrans}\ p, \mathit{pclNextState}\ p)\ e_{ctrl\ step} = \mathbf{true} \tag{6.43}$$

We prove the following conjunct of $PclStep$. The remaining ones are trivial to prove.

$$\exists\, p \in Parcels.\ p \cap \mathit{RegPcl} \subseteq \mathit{dom}\ q_{PAc\,1} \wedge \mathit{roots}\ ((\mathit{pclTrans}\ p).\mathit{fg}) = p$$

We have $p \cap \mathit{RegPcl} \subseteq \mathit{dom}\ q_{PAc\,1}$ since $\mathit{pclState}\ p \subseteq q_{PAc\,1}$.

Equation 6.43 together with Equation 6.32 in the inductive definition of $pa_{c\,1}$ imply

$$(q_{PAc\,1}, \mathit{pclTrans}\ p, \mathit{pclNextState}\ p) \in R_{PAc\,1} \tag{6.44}$$

Noting that $\mathit{roots}\ ((\mathit{pclTrans}\ p).\mathit{fg}) \cap \mathit{RegPcl} = \mathit{dom}\ (\mathit{pclState}\ p)$ and since $\overline{pa}_{c\,1}$ is the closure of

$pa_{c\,1}$, Equation 6.44 implies

$$(\mathit{pclState}\ p, \mathit{pclTrans}\ p, \mathit{pclNextState}\ p) \in \overline{R}_{PAc\,1}$$

Part 2

$$\begin{array}{l} \forall\, (q'_{Pc}, t'_{Pc}, q''_{Pc}) \in R_{Pc}.\ \forall\, p' \in PclMap\ (q'_{Pc}, t'_{Pc}, q''_{Pc}). \\ \quad \exists\, q_{PAc\,1}{}' \in Q_{PAc\,1}.\ \exists\, q_{ctrl}{}' \in \mathit{PEnv}(\mathit{RegCtrl}). \\ \qquad (q_{PAc\,1}{}' \subseteq q_{Pc}{}') \wedge (q_{ctrl}{}' \subseteq q_{Pc}{}') \wedge (\mathit{pclState}\ p' \subseteq q_{PAc\,1}{}') \\ \qquad\quad \wedge\, (q_{PAc\,1}{}', q_{ctrl}{}') \in PclStatePairs \end{array} \tag{6.45}$$

To prove Equation 6.45 we use the continuity property (Equation 5.3 on page 101) in the definition of the parcel map. Consider $(q'_{Pc}, t'_{Pc}, q''_{Pc}) \in R_{Pc}$ and $p' \in PclMap\ (q'_{Pc}, t'_{Pc}, q''_{Pc})$. Therefore, there exists a parcel p at step $(q_{Pc}, t_{Pc}, q'_{Pc})$ so that $\mathit{pclState}\ p' \subseteq \mathit{pclNextState}\ p$. Using the induction hypothesis, there exist $q_{PAc\,1}$ and q_{ctrl} such that:

$$\begin{array}{rcl} q_{PAc\,1} & \subseteq & q_{Pc} \\ q_{ctrl} & \subseteq & q_{Pc} \\ (q_{PAc\,1}, q_{ctrl}) & \in & PclStatePairs \end{array}$$

We perform the construction in Equation 6.40 to Equation 6.42. Equation 6.43 together with Equation 6.32 in the inductive definition of $PclStatePairs$ imply that $(\mathit{pclNextState}\ p, q_{ctrl}{}') \in PclStatePairs$. We therefore set $q_{PAc\,1}{}' = \mathit{pclNextState}\ p$.

□

6.2.3 Proposition. If $\overline{I}_{PAc\,1} \subseteq I_{PAc}$ then

$$Pipe_c \left[\mathit{Dps}\ \overline{pa}_{c\,1} / Dps_c\right] \preceq_P Pipe_c$$

Proof. The parcel automaton $pa_{c\,1}$ differs from the induced parcel automaton by representing unreachable datapath behaviours. Its closure does not add new behaviours. Unreachable behaviours are already specified by the datapath circuits and therefore do not change the datapath when performing abstract interpretation. □

6.3 Parcel Steps In Propositional Logic

A path through the concrete parcel automaton consists of a sequence of parcel transitions. Since the parcel automata we work with are closed we consider only parcel transitions that are parcel steps. Each such step satisfies the predicate $PclStep$ described in Equation 6.29. The essential part of the abstraction algorithm is to derive the equivalence classes of the parcel steps $(q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc})$ from a constrained state q^k_{PAc}, where the constraint requires that the state q^k_{PAc} be reachable by a path $\pi^k_{PA\,c}$ equivalent to the abstract path $\pi^k_{PA\,a}$.

In order to define the representation of $PclStep$ in proposition logic, we observe that the span of a parcel step $(q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc})$ (i.e. the parcel variables, datapaths and control variables that it mentions), is limited by the domain of q^k_{PAc} and by the set of all possible parcels $Parcels$. We denote by $\tilde{p}^k$ our conservative approximation of a possible parcel in step $(q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc})$:

$$\tilde{p}^k \equiv \bigcup \{\, p \mid p \in Parcels \wedge p \cap \mathit{RegPcl} \subseteq \mathit{dom}\, q^k_{PAc} \,\} \tag{6.46}$$

The cone of influence of a set of pipeline variables V is a set of pipeline variables $\mathit{cone}\, V$ that consists of the variables that can be assigned transitively from V.

6.3.1 Definition (Cone Of Influence)**.** The cone of influence of a parcel is the smallest set satisfying:

Base Case $V \subseteq \mathit{cone}\, V$

Inductive Case

- If $v_l \in \mathit{cone}\, V$ and there exists '$v_k := e$' $\in Pipe_c.Tr$ such that $v_l \in \mathit{vars}\, e$ then $v_k \in \mathit{cone}\, V$.
- If $dp \in Dps$ and $\mathit{Arg}(dp.\mathit{PclP}) \cap (\mathit{cone}\, V) \neq \emptyset$ then $\mathit{Arg}(dp.\mathit{PclN}) \cup \mathit{OutputArg}(dp.V_{ctrl}) \subseteq \mathit{cone}\, V$.

We extend the notation and write $dp \in \mathit{cone}\, p$ if any of the input parcel parameters of the datapath dp is in the cone of the parcel, i.e. $\mathit{Arg}(dp.\mathit{PclP}) \cap (\mathit{cone}\, p) \neq \emptyset$. Similarly, for fan-out edges we write $(v_l, b, v_k) \in \mathit{cone}\, p$ if $v_l \in \mathit{cone}\, p$.

The parcel at step k is denoted by $p^k \subseteq \tilde{p}^k$. The representation of $PclStep$ in propositional logic requires that variables encode the current state, transition label and next state of parcel p^k. In addition, we also need to represent the associated current and next control states and the fan-out graph of the parcel. Consistency with the datapath behaviour is specified by the propositional formula $[R_{dp}]_{bool}$ that represents the transition relation for each datapath $dp \in \mathit{cone}\, \tilde{p}^k$. We denote the set of variables corresponding to a datapath instance by V_{dp}.

The variables that make up the parcel step variables $\widetilde{V_{step}}$ are defined below:

$$\begin{aligned}
\widetilde{\mathit{RegPcl}} &\equiv \mathit{dom}\, q^k_{PAc} \\
\widetilde{\mathit{CombPcl}} &\equiv \mathit{CombPcl} \cap \mathit{cone}\, \tilde{p}^k \\
\widetilde{\mathit{NextRegPcl}} &\equiv \mathit{NextRegPcl} \cap \mathit{cone}\, \tilde{p}^k \\
\widetilde{\mathit{ControlVars}} &\equiv \bigcup_{t^k_{PAc}} \mathit{ControlEnvDom}\,(q^k_{PAc}, q^k_{ctrl}, t^k_{PAc}) \\
\widetilde{V_{Dps}} &\equiv \bigcup_{dp \in \mathit{cone}\, \tilde{p}^k} V_{dp} \\
\widetilde{\mathit{Parcel}} &\equiv \{\, pcl_v \mid v \in \tilde{p}^k \,\} \\
\widetilde{V_{fanOut}} &\equiv \{\, fanOut_v \mid v \in (\mathit{cone}\, \tilde{p}^k) \cap (V_{pcl} \cup \mathit{NextRegPcl}) \,\}
\end{aligned}$$

where

- $\widetilde{\mathit{RegPcl}}$ represents the current state of the parcel.
- $\widetilde{\mathit{CombPcl}}$ denotes the combinational parcel variables in the cone.
- $\widetilde{\mathit{NextRegPcl}}$ are the next state parcel registers in the cone of $\tilde{p}^k$.
- $\widetilde{\mathit{ControlVars}}$ denotes the upper bound on the domain of possible environments $e^k_{ctrl\ step}$. This is chosen heuristically.
- The set of datapath instance variables in the cone of the parcel is denoted by $\widetilde{V_{Dps}}$.
- The set of variables $\widetilde{\mathit{Parcel}}$ encode the parcel p^k as a subset of $\tilde{p}^k$. For $v \in \tilde{p}^k$ we have that $pcl_v = \mathbf{true}$ if the parcel p^k contains v.
- The set of fan-out variables $\widetilde{V_{fanOut}}$ represent whether $v \in (\mathit{cone}\, \tilde{p}^k) \cap (V_{pcl} \cup \mathit{NextRegPcl})$ is in the fan-out of the parcel p^k. Therefore $fanOut_v = \mathbf{true}$ if $v \in (p^k)^*$.

We define

$$\widetilde{\mathit{NextRegPcl}}^{k+1} \equiv \left(\ \widetilde{\mathit{NextRegPcl}}\left[{}^{\mathit{RegPcl}}/_{\mathit{NextRegPcl}}\right]\ \right)^{k+1}$$

With the above above sets of variables the set of parcel step variables is described as follows:

$$\widetilde{V_{step}}^k = \widetilde{\mathit{RegPcl}}^k \uplus \widetilde{\mathit{CombPcl}}^k \uplus \widetilde{\mathit{NextRegPcl}}^{k+1} \uplus \widetilde{\mathit{ControlVars}}^k \uplus \widetilde{V_{Dps}}^k \uplus \widetilde{\mathit{Parcel}}^k \uplus \widetilde{V_{fanOut}}^k$$

We are now prepared to describe the formula $[\![\mathit{PclStep}]\!]_{bool}\,(\widetilde{V_{step}}^k)$.

$$[\![PclStep]\!]_{bool}\,(\widetilde{V_{step}}^{k}) \equiv \left(\begin{array}{c} [\![Assignments]\!]_{bool} \\ \wedge \\ [\![Datapaths]\!]_{bool} \\ \wedge \\ [\![FanOutPropagation]\!]_{bool} \\ \wedge \\ [\![WellDefinedPclStep]\!]_{bool} \end{array} \right) \tag{6.47}$$

The description of $PclStep$ in propositional logic consists of four subformulas. The variables that could occur in the parcel step are constrained by the relevant assignments and datapaths of the pipeline model. The formula in Equation 6.48 consists of the assignments to parcel and control variables in the cone of the parcel. The formula in Equation 6.49 ensures the consistency of the parcel step with respect to datapath behaviour.

$$[\![Assignments]\!]_{bool} \equiv \left[\!\!\left[(Pipe_c.C.Tr) \mid \textit{cone}\ \tilde{p}^k \right]\!\!\right]_{bool} (\widetilde{V_{step}}^{k}) \tag{6.48}$$

$$[\![Datapaths]\!]_{bool} \equiv \bigwedge_{dp \in \textit{cone}\ \tilde{p}^k} [\![R_{dp}]\!]_{bool}\,(\widetilde{V_{step}}^{k}) \tag{6.49}$$

The following formula propagates the fan-out of the parcel through the fan-out edges in the cone. The first conjunct represents the base case: the variables that are part of the parcel are also in its fan-out. The second conjunct corresponds to the inductive case, as the fan-out propagates transitively. The third case covers the case of parcel variables that are not derived transitively from the parcel.

$$[\![FanOutPropagation]\!]_{bool} \equiv \left(\begin{array}{c} \bigwedge\limits_{v \in \tilde{p}^k} pcl_v^k \Longrightarrow fanOut_v^k \\ \wedge \\ \bigwedge\limits_{(v_1,b,v_2) \in \textit{cone}\ \tilde{p}^k} b \Longrightarrow (fanOut_{v_1}^k = fanOut_{v_2}^k) \\ \wedge \\ \bigwedge\limits_{v_2 \in \textit{cone}\ \tilde{p}^k} \left(\neg pcl_{v_2}^k \wedge \left(\bigwedge\limits_{(v_1,b,v_2) \in \textit{cone}\ \tilde{p}^k} \neg b \right) \Longrightarrow \neg fanOut_{v_2}^k \right) \end{array} \right) \tag{6.50}$$

The last subformula ensures the parcel step is well defined. A datapath must have either all arguments in the parcel's fan-out or none at all.

$$[\![WellDefinedPclStep]\!]_{bool} \equiv \bigwedge_{dp \in \textit{cone}\ \tilde{p}^k} \left(\bigwedge_{\{\, pclP_1, pclP_2 \,\} \subseteq \textit{Arg}(dp.\textit{PclP})} fanOut_{pclP_1}^k = fanOut_{pclP_2}^k \right) \tag{6.51}$$

Finally, we provide the semantics of our encoding by defining the satisfiability of $[\![PclStep]\!]_{bool}\,(\widetilde{V_{step}}^{\,k})$ by a parcel step. We use the notation $t^k_{PAc} = \langle fg^k, e^k_{ctrl\,c}, e^k_{pcl\,c}\rangle$.

6.3.2 Definition (Propositional Semantics Of Parcel Steps)**.** We write

$$(q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc}) \models [\![PclStep]\!]_{bool}\,(\widetilde{V_{step}}^{\,k})$$

if there exists an environment $e^k_{step} \in \mathit{Env}(\widetilde{V_{step}}^{\,k})$ such that:

$$\begin{aligned}
e^k_{step} &\models [\![PclStep]\!]_{bool}\,(\widetilde{V_{step}}^{\,k}) \\
q^k_{PAc}\left[\widetilde{\mathit{RegPcl}}^{\,k}/\mathit{RegPcl}\right] &\subseteq e^k_{step} && (6.52)\\
e^k_{pcl\,c}\left[\widetilde{\mathit{CombPcl}}^{\,k}/\mathit{CombPcl}\right] &\subseteq e^k_{step} && (6.53)\\
e^k_{ctrl\,c}\left[\widetilde{\mathit{ControlVars}}^{\,k}/\widetilde{\mathit{ControlVars}}\right] &\subseteq e^k_{step} && (6.54)\\
q^{k+1}_{PAc}\left[\widetilde{\mathit{NextRegPcl}}^{\,k}/\mathit{RegPcl}\right] &\subseteq e^k_{step} && (6.55)\\
e^k_{step}(pcl^k_v) &= \begin{cases}\textbf{true} & : v \in \mathit{roots}\ fg^k\\ \textbf{false} & : v \notin \mathit{roots}\ fg^k\end{cases} && (6.56)\\
e^k_{step}(fanOut^k_v) &= \begin{cases}\textbf{true} & : v \in fg^k.\mathit{Succ}\\ \textbf{false} & : v \notin fg^k.\mathit{Succ}\end{cases} && (6.57)
\end{aligned}$$

Proposition 6.3.3 states the correctness of our encoding.

6.3.3 Proposition.

$$\Big(\ (q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc}) \models [\![PclStep]\!]_{bool}\,(\widetilde{V_{step}}^{\,k})\ \Big) \Longleftrightarrow \exists e^k_{ctrl\,step}.\ PclStep\ (q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc})\ e^k_{ctrl\,step} \tag{6.58}$$

Given an environment e^k_{step} that satisfies $[\![PclStep]\!]_{bool}\,(\widetilde{V_{step}}^{\,k})$ we can define the corresponding parcel step $(q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc})$ so that $(q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc}) \models [\![PclStep]\!]_{bool}\,(\widetilde{V_{step}}^{\,k})$.

$$\begin{aligned}
q^k_{PAc} &= (e^k_{step} \mid \widetilde{\mathit{RegPcl}}^{\,k})\left[\mathit{RegPcl}/\widetilde{\mathit{RegPcl}}^{\,k}\right] && (6.59)\\
fg^k.\mathit{Nodes} &= \{\, v \mid e^k_{step}(fanOut^k_v) = \textbf{true}\,\} && (6.60)\\
fg^k.\mathit{Succ} &= \{\, (v_1, b, v_2) \in \mathit{FanOutEdges} \mid
\end{aligned}$$

$$(fanOut^k_{v_1} = \mathbf{true}) \wedge (fanOut^k_{v_2} = \mathbf{true}) \wedge e^k_{step} \models b\,\} \tag{6.61}$$

$$e^k_{pcl\,c} = (e^k_{step} \mid (fg^k.\mathit{Nodes} \cap \mathit{CombPcl})^k) \left[\widetilde{\mathit{CombPcl}}/\widetilde{\mathit{CombPcl}}^k\right] \tag{6.62}$$

$$e^k_{ctrl\,c} = \left(e^k_{step} \mid \left(\bigcup_{dp \in \mathit{datapaths}\ fg^k} \mathit{Arg}(dp.V_{ctrl}) \right)^k \right) \left[\widetilde{\mathit{ControlVars}}/\widetilde{\mathit{ControlVars}}^k\right] \tag{6.63}$$

$$q^{k+1}_{PAc} = (e^k_{step} \mid \widetilde{\mathit{NextRegPcl}}^k) \left[\mathit{RegPcl}/\widetilde{\mathit{NextRegPcl}}^k\right] \tag{6.64}$$

The step formula f^k_{step} stands for an equivalence class of parcel steps $\left[\!\left[(q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc})\right]\!\right]_{=_{PA}}$. It consists of two parts, one subformula for the parcel step and another for that encodes the equivalence class of the transition label t^k_{PAc}:

$$f^k_{step}(\widetilde{V_{step}}^k) \equiv \left[\!\left[\left[\!\left[(q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc})\right]\!\right]_{=_{PA}}\right]\!\right]_{bool}(\widetilde{V_{step}}^k)$$

$$\left[\!\left[\left[\!\left[(q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc})\right]\!\right]_{=_{PA}}\right]\!\right]_{bool}(\widetilde{V_{step}}^k) \equiv [\![\mathit{PclStep}]\!]_{bool}(\widetilde{V_{step}}^k) \wedge \left[\!\left[\left[\!\left[t^k_{PAc}\right]\!\right]_{=_{PA}}\right]\!\right]_{bool}(\widetilde{V_{step}}^k)$$

$$\left[\!\left[\left[\!\left[t^k_{PAc}\right]\!\right]_{=_{PA}}\right]\!\right]_{bool}(\widetilde{V_{step}}^k) \equiv \left(\bigwedge_{(e,b,v_2) \in fg^k.\mathit{Succ}} b \;\wedge\; \bigwedge_{(e,b,v_2) \in \mathit{cone}\ \tilde{p}^k \setminus fg^k.\mathit{Succ}} \neg b \right) \wedge \bigwedge_{v \in \mathit{dom}\ e^k_{ctrl\,c}} v = e^k_{ctrl\,c}(v)$$

The encoding of the equivalence class of the transition label t^k_{PAc} consists of two parts, corresponding to the fan-out graph fg^k and respectively, to the control environment $e^k_{ctrl\,c}$.

6.3.4 Definition (Propositional Semantics of Equivalence Classes Of Parcel Steps)**.** We write

$$(q^k_{PAc\,1}, t^k_{PAc\,1}, q^{k+1}_{PAc\,1}) \models f^k_{step}(\widetilde{V_{step}}^k)$$

if there exists an environment $e^k_{step\,c\,1} \in \mathit{Env}(\widetilde{V_{step}}^k)$ satisfying Equation 6.52 up to Equation 6.57 and

$$e^k_{step\,c\,1} \models f^k_{step}(\widetilde{V_{step}}^k)$$

6.3.5 Proposition. Step formulas correspond to equivalence classes of parcel steps.

$$(q^k_{PAc\,1}, t^k_{PAc\,1}, q^{k+1}_{PAc\,1}) \in \left[\!\left[(q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc})\right]\!\right]_{=_{PA}} \Longleftrightarrow (q^k_{PAc\,1}, t^k_{PAc\,1}, q^{k+1}_{PAc\,1}) \models f^k_{step}$$

6.4 Path Formulas

Path formulas are propositional formulas that represent equivalence classes of concrete paths in the parcel automaton $pa_{c\,1}$. A path formula is satisfiable if and only if it stands for such an equivalence class. A path formula f^k_{path} corresponds to paths $\pi^k_{PA\,c}$ of length k. The path formula describes each step i of the path using a step formula f^i_{step}. Corresponding to the inductive definition of a path

$$\begin{aligned} \pi^{0}_{PA\,c} &= q^0_{PAc} \\ \pi^{k+1}_{PA\,c} &= \pi^k_{PA\,c} \xrightarrow{t^k_{PAc}} q^{k+1}_{PAc} \end{aligned}$$

we have a similar type of definition of path formulas:

$$\begin{aligned} f^0_{path} &= \textbf{true} \\ f^{k+1}_{path} &= f^k_{path} \wedge f^k_{step} \end{aligned}$$

6.4.1 Definition (Propositional Semantics Of Parcel Automaton Paths)**.** We define

$$\pi^k_{PA\,c} \models f^k_{path}(\widetilde{V_{step}}^k)$$

by induction:

Base Case For paths of length 0 we have:

$$\pi^{0}_{PA\,c} \models \mathbf{true}$$

Inductive Case

$$\begin{aligned} &\pi^{k+1}_{PA\,c} \models f^{k+1}_{path} \\ &\quad\equiv \\ &\qquad \pi^k_{PA\,c} \models f^k_{path} \wedge (q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc}) \models f^k_{step} \end{aligned}$$

6.4.2 Proposition. Step formulas stand for equivalence classes of parcel automaton paths.

$$\pi^k_{PA\,c\,1} \in \left[\!\left[\pi^k_{PA\,c}\right]\!\right]_{=_{PA}} \Longleftrightarrow \pi^k_{PA\,c\,1} \models f^k_{path}$$

The parcel automaton $pa_{c\,1}$ described in Section 6.2 is defined inductively. The inductive definition is exploited in the use of path formulas which are used to explore the control-visible datapath behaviours represented by the parcel automaton. The control state associated with the parcel state is propagated along the path formula by sharing variables between consecutive steps.

The set $\widetilde{ControlVars}$ represents the current and next control states as follows:

$$\begin{aligned} \mathit{dom}\, q_{ctrl} &\subseteq \widetilde{ControlVars} \\ (\mathit{dom}\, q_{ctrl}{}')\left[\mathit{NextRegCtrl}/\mathit{RegCtrl}\right] &\subseteq \widetilde{ControlVars} \end{aligned}$$

In the k-th step this corresponds to

$$\begin{aligned} \mathit{dom}\, q^k_{ctrl}\left[\widetilde{\mathit{RegCtrl}}^k/\mathit{RegCtrl}\right] &\subseteq \widetilde{ControlVars}^k \\ (\mathit{dom}\, q^{k+1}_{ctrl})\left[\widetilde{\mathit{NextRegCtrl}}^k/\mathit{RegCtrl}\right] &\subseteq \widetilde{ControlVars}^k \end{aligned}$$

To propagate the control state q^{k+1}_{ctrl} to the $k+1$-th step, we perform variable substitution in the step formula f^k_{step}:

$$\begin{aligned} A &\equiv (\mathit{dom}\, q^{k+1}_{ctrl})\left[\widetilde{\mathit{NextRegCtrl}}^k/\mathit{RegCtrl}\right] \\ B &\equiv A\left[\widetilde{\mathit{RegCtrl}}^{k+1}/\widetilde{\mathit{NextRegCtrl}}^k\right] \\ \widetilde{f^k_{step}} &\equiv f^k_{step}\left[A/B\right] \end{aligned} \tag{6.65}$$

6.5 Abstraction Algorithms

The basic abstraction algorithm constructs an abstract parcel automaton state for each equivalence class of the set of paths $\Pi(pa_{c\,1})$ with respect to the equivalence on paths '$=_{PA}$'. The abstract initial states correspond to paths of length one, that consist of a single state. The equivalence classes of such paths correspond to sets of initial states that have the same domain. The set of domains of initial states is denoted by $InitParcels$.

$$InitParcels \equiv \{\, \mathit{dom}\, q_{PAc} \mid q_{PAc} \in I_{PAc\,1} \,\}$$

Accordingly, the set of abstract initial states I_{PAa} is in bijection with the set of domains of the concrete initial states:

$$I_{PAa} \leftrightarrow InitParcels \tag{6.66}$$

At the current step the algorithm extends the abstract path

$$\pi^k_{PA\,a} = q^0_{PAa} \xrightarrow{t^0_{PAa}} \cdots \xrightarrow{t^{k-1}_{PAa}} q^k_{PAa} \tag{6.67}$$

The path $\pi^k_{PA\,a}$ stands for the non-empty set of concrete states $CurrentStates$ reachable along equivalent concrete paths:

$$CurrentStates \equiv \{\, q_{PAc} \mid \exists\, \pi^k_{PA\,c} \in \Pi(\overline{pa}_{c\,1}).\ \overline{I}_{PAc\,1} \overset{\pi^k_{PA\,c}}{\rightsquigarrow} q_{PAc} \wedge (\pi^k_{PA\,c} =_{PA} \pi^k_{PA\,a}) \,\} \tag{6.68}$$

The equivalence class of concrete states that $\pi^k_{PA\,a}$ denotes is represented by the path formula f^k_{path}. At any given time, f^k_{path} is satisfiable.

The algorithm finds all possible extensions $\pi^{k+1}_{PA\,c\,1}$ of the paths $\pi^k_{PA\,c}$ equivalent to $\pi^k_{PA\,a}$. For each equivalence class of the set of paths $\pi^{k+1}_{PA\,c\,1}$ the algorithm constructs an abstract state q^{k+1}_{PAa} and the path $\pi^{k+1}_{PA\,a}$ that continues $\pi^k_{PA\,a}$. The correspondence is shown by the following commuting diagram:

$$\begin{array}{ccccc}
 & & \overbrace{\phantom{I_{PAa} \xrightarrow{\pi^k_{PA\,a}} q^k_{PAa} \xrightarrow{t^k_{PAa}} q^{k+1}_{PAa}}}^{\pi^{k+1}_{PA\,a}} & & \\
I_{PAa} & \xrightarrow{\pi^k_{PA\,a}} & q^k_{PAa} & \xrightarrow{t^k_{PAa}} & q^{k+1}_{PAa} \\
 & & \uparrow{\scriptstyle \mathcal{S}_{PA}} & & \vdots{\scriptstyle \mathcal{S}_{PA}} \\
I_{PAc} & \xrightarrow{\pi^k_{PA\,c}} & q^k_{PAc\,1} & \xrightarrow{t^k_{PAc\,1}} & q^{k+1}_{PAc\,1} \\
 & & \underbrace{\phantom{I_{PAa} \xrightarrow{\pi^k_{PA\,a}} q^k_{PAa} \xrightarrow{t^k_{PAa}} q^{k+1}_{PAa}}}_{\pi^{k+1}_{PA\,c\,1}} & &
\end{array} \tag{6.69}$$

Concrete paths $\pi^{k+1}_{PA\,c\,1}$ of length $k+1$ extend paths of length k that are equivalent to $\pi^k_{PA\,a}$ if

$$\pi^{k+1}_{PA\,c\,1} \models f^k_{path} \wedge [\![PclStep]\!]_{bool}\,(\widetilde{V_{step}}^{\,k}) \tag{6.70}$$

Equation 6.70 is the basis of the path extension algorithm in Algorithm 6.1.

Each iteration of the while loop in Algorithm 6.1 solves a constrained path formula of form $f^k_{path} \wedge Constraint$. Initially, $Constraint = [\![PclStep]\!]_{bool}\,(\widetilde{V_{step}}^{\,k})$. The solution to the path formula is interpreted as a concrete step $(q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc})$ (lines 5–6). At the end of the iteration, the equivalence class corresponding to the current solution is excluded from future solutions by adding the negation $\neg \left[\!\left[[\![t^k_{PAc}]\!]_{=_{PA}} \right]\!\right]_{bool}$ to the constrained path formula (line 19).

Corresponding to the concrete step $(q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc})$, the algorithm constructs t^k_{PAa} and q^{k+1}_{PAa}. In order to make the diagram in Equation 6.69 commute we must have

$$\begin{array}{rcl}
t^k_{PAc} & = & \langle fg^k_c, e^k_{ctrl\,c}, e^k_{pcl\,c} \rangle \\
t^k_{PAa} & = & \langle fg^k_a, e^k_{ctrl\,a}, e^k_{pcl\,a} \rangle \\
fg^k_c & \approx_{ai} & fg^k_a \text{ (equality modulo constants)}
\end{array}$$

Input: $\langle q^k_{PAa}, f^k_{path} \rangle$

Output: Abstract steps: $(q^k_{PAa}, t^k_{PAa\,1}, q^{k+1}_{PAa\,1}), (q^k_{PAa}, t^k_{PAa\,2}, q^{k+1}_{PAa\,2}), \ldots$

1 $f := f^k_{path} \wedge [\![PclStep]\!]_{bool}\,(\widetilde{V_{step}}^k)$
2 $result := \emptyset$
3 **while** f *is satisfiable* **do**
4 $\quad e_{path\,c} := \mathit{solve}(f)$
5 $\quad e^k_{step} := e_{path\,c} \mid \widetilde{V_{step}}^k$
$\quad$ `// Use construction in Equation 6.59 up to Equation 6.64`
6 $\quad (q^k_{PAc}, t^k_{PAc}, q^{k+1}_{PAc}) := \mathit{step}(e^k_{step})$
7 $\quad \langle fg^k_c, e^k_{ctrl\,c}, e^k_{pcl\,c} \rangle := t^k_{PAc}$
8 $\quad p^k := \mathit{parcel}(e^k_{step} \mid \widetilde{Parcel}^k)$
9 $\quad e^k_{ctrl\,a} := e^k_{ctrl\,c}$ `// must be equal`
10 $\quad e^k_{pcl\,a\,1} := \mathit{createAbstractValues}(\langle p^k, fg^k_c \rangle)$
11 $\quad e^k_{pcl\,a} := e^k_{pcl\,a\,1} \mid_{V_c}$
$\quad$ `//` $fg^k_a \approx_{ai} fg^k_c$
12 $\quad t^k_{PAa} := \langle fg^k_a, e^k_{ctrl\,a}, e^k_{pcl\,a} \rangle$
13 $\quad$ **if** $e^k_{pcl\,a\,1} \mid_{\mathit{NextRegPcl}} \neq \emptyset$ **then**
14 $\quad\quad q^{k+1}_{PAa} := (e^k_{pcl\,a\,1} \mid_{\mathit{NextRegPcl}}) \left[\mathit{RegPcl}/\mathit{NextRegPcl}\right]$
15 $\quad$ **else**
16 $\quad\quad q^{k+1}_{PAa} := \mathit{final}_{PAa}$
17 $\quad$ **end**
18 $\quad$ append $(q^k_{PAa}, t^k_{PAa}, q^{k+1}_{PAa})$ to $result$
19 $\quad f := f \wedge \neg \left[\!\!\left[[\![t^k_{PAc}]\!]_{=_{PA}} \right]\!\!\right]_{bool}$
20 **end**

Algorithm 6.1: Path Extension Algorithm

$$e^k_{ctrl\,a} \quad = \quad e^k_{ctrl\,c}$$

We define q^{k+1}_{PAa} over the next-state registers that appear in the fan-out graph fg:

$$\mathit{dom}\, q^{k+1}_{PAa} = \{\, v \mid v' \in fg.\mathit{Nodes} \,\}$$

It remains to describe how $e^k_{pcl\,a}$ and q^{k+1}_{PAa} evaluate the combinational variables and, respectively, the next-state parcel variables (lines 2–19) of Algorithm 6.2. The edges of the fan-out graph fg^k_c describe value copying or datapath transformations. For each parameter of a datapath output parcel variable $pclN$ we create a new abstract value that denotes the corresponding transformation (line 10). Similarly, a new value is used for inputs (line 3) or **choice** assigned variables (line 18). For each constant variable v we choose an advance an abstract value w_a which we assign to v at

Input: $\langle p^k, fg_c^k \rangle$
Output: Abstract parcel environment: $e^k_{pcl\ a\ 1}$

1 $e^k_{pcl\ a\ 1} := \emptyset$ // $e^k_{pcl\ a\ 1}$ stands for $t^k_{PAa} \cup q^{k+1}_{PAa}$
2 **foreach** $v \in$ *InputPcl* $\cap\ p^k$ **do**
3 $e^k_{pcl\ a\ 1}(v) :=$ *newAbstractValue*()
4 **end**
5 **foreach** $(v_1, b, v_2) \in fg_c^k.$*Succ* **do**
6 mark v_2
7 **if** $v_1 \notin$ *PclN* **then**
8 $e^k_{pcl\ a\ 1}(v_2) := (e^k_{pcl\ a} \cup q^k_{PAa})(v_1)$
9 **else**
10 $e^k_{pcl\ a\ 1}(v_2) :=$ *newAbstractValue*()
11 **end**
12 **end**
13 **foreach** $v \in p^k \cap V_c$ **do**
14 **if** v *is unmarked* **then**
15 **if** v *is assigned a constant in* fg_c^k **then**
16 $e^k_{pcl\ a\ 1}(v) := w_a$ // w_a is the abstract constant for the edge $(w_c, b, v) \in fg_c^k$
17 **else**
18 $e^k_{pcl\ a\ 1}(v) :=$ *newAbstractValue*()// v is assigned choice
19 **end**
20 **end**
21 **end**

Algorithm 6.2: Abstract Value Propagation

line 16.

Formally, we define the environment $e^k_{pcl\ a\ 1} \in$ *Env*(fg.*Nodes*) that stands for $t^k_{PAa} \cup q^{k+1}_{PAa}$ by induction:

Base Case

- If $v \in$ *RegPcl* then $e^k_{pcl\ a\ 1}(v) = q^k_{PAa}(v)$.
- If $v \in$ *InputPcl* then $e^k_{pcl\ a\ 1}(v) =$ *newAbstractValue*()
- If v is assigned a constant w_c then $e^k_{pcl\ a\ 1}(v) = w_a$, where w_a is the abstract constant corresponding to the edge $(w_c, b, v) \in fg_c^k$.
- Otherwise, v is assigned **choice**, so $e^k_{pcl\ a\ 1}(v) =$ *newAbstractValue*().

Inductive Case

- If $v \notin$ *PclN* then there exists $(v_l, b, v_k) \in fg$.*Succ* so that $v = v_k$. We set $e^k_{pcl\ a\ 1}(v) = e^k_{pcl\ a\ 1}(v_l)$.

- If $v \in \mathit{PclN}$ then $v = \mathit{newAbstractValue}()$.

Using $e^k_{\mathit{pcl}\ a\ 1}\, k$ we define:

$$e_{\mathit{pcl}\, a} = e^k_{\mathit{pcl}\ a\ 1} \mid_{\mathit{CombPcl}} \tag{6.71}$$

$$q^{k+1}_{\mathit{PA}\, a} = \begin{cases} (e^k_{\mathit{pcl}\ a\ 1} \mid_{\mathit{NextRegPcl}}) \left[{}^{\mathit{NextRegPcl}}/_{\mathit{RegPcl}}\right] & : e^k_{\mathit{pcl}\ a\ 1} \mid_{\mathit{NextRegPcl}} \neq \emptyset \\ \mathit{final}_{\mathit{PA}\, a} & : e^k_{\mathit{pcl}\ a\ 1} \mid_{\mathit{NextRegPcl}} = \emptyset \end{cases} \tag{6.72}$$

The abstraction algorithm is represented in Algorithm 6.3. It performs a depth-first-search exploration of the paths in the concrete parcel automaton $pa_{c\,1}$. The algorithm maintains a stack of concrete paths to be explored. The concrete path $\pi^k_{\mathit{PA}\, c}$ that is currently explored is represented by the formula f^k_{path} such that $\pi^k_{\mathit{PA}\, c} \models f^k_{path}$. The entries of the stack consist of tuples $\langle q^k_{\mathit{PA}\, a}, dom^k_{ctrl}, f^k_{path}\rangle$, where the abstract $q^k_{\mathit{PA}\, a}$ is on the frontier of the on-the-fly construction of the abstract parcel automaton. It is reached in the abstract parcel automaton by a path $\pi^k_{\mathit{PA}\ a}$ equivalent to $\pi^k_{\mathit{PA}\, c}$. For each possible extension of the path $\pi^k_{\mathit{PA}\, c}$ of form $(q^k_{\mathit{PA}\, c}, t^k_{\mathit{PA}\, c}, q^{k+1}_{\mathit{PA}\, c})$, the algorithm creates a new equivalent transition in the parcel automaton (lines 19–29).

The test at line 23 fails if the stack contains an already visited abstract state $q_{\mathit{PA}\, a}$ that subsumes the newly constructed state $q^{k+1}_{\mathit{PA}\, a}$. This check ensures termination of the algorithm when the runs of $pa_{c\,1}$ are terminating (Definition 6.1.2). If the abstract state $q^{k+1}_{\mathit{PA}\, a}$ has not been visited before, a new entry is added to the stack. It is possible the state $q^{k+1}_{\mathit{PA}\, a}$ has been visited before, but it is not on the stack. In the DFS algorithm $(q^k_{\mathit{PA}\, a}, t^k_{\mathit{PA}\, a}, q^{k+1}_{\mathit{PA}\, a})$ is called a cross edge. To be sound, the algorithm must visit $q^{k+1}_{\mathit{PA}\, a}$ under the current path constraint.

The abstraction algorithm explores the paths of the concrete parcel automaton $pa_{c\,1}$ according to the inductive definition of $pa_{c\,1}$ on page 150. At line 23 the abstraction algorithm checks whether the pair of abstract state, control state has been visited on the current path. The termination of the algorithm is proven if the heuristic function $\mathit{ControlEnvDom}$ has the property that for transitions that only copy the parcel state, the next control state is empty. This requirement is justifiable since the purpose of of the next-state control variables in $\mathit{ControlEnvDom}$ is to prevent unreachable computations due to datapath transformations that propagate into the parcel's next state.

The proof of correctness of Algorithm 6.3 is based on showing that the algorithm performs path abstraction of the concrete parcel automaton $\overline{pa}_{c\,1}$. From Lemma 6.1.1 we get

$$\begin{array}{rcl} \mathcal{L}(\overline{pa}_{c\,1}) & \subseteq & \mathcal{L}(pa_a) \\ \overline{pa}_{c\,1} & \preceq_{\mathit{PA}} & pa_a \end{array}$$

We begin by showing termination. We show that the algorithm explores a finite number of equiva-

Input: $Pipe_c$
Output: $\langle Q_{PAa}, R_{PAa}, T_{PAa}, I_{PAa} \rangle$

$\langle Q_{PAa}, R_{PAa}, T_{PAa}, I_{PAa} \rangle := \langle \{ \mathit{final}_{PAa} \}, \emptyset, \emptyset, \emptyset \rangle$
$Stack := \emptyset$
$Visited := \emptyset$
foreach $p \in InitParcels$ **do**
    $q^0_{PAa} :=$ *newAbstractState*(p)
    $I_{PAa} := I_{PAa} \cup \{ q^0_{PAa} \}$
    $Q_{PAa} := Q_{PAa} \cup \{ q^0_{PAa} \}$
    $f^0_{path} := \mathbf{true}$
    $dom^0_{ctrl} := \emptyset$
    push $Stack$, $\langle q^0_{PAa}, dom^0_{ctrl}, \mathbf{true} \rangle$
end
while $Stack \neq \emptyset$ **do**
    $\langle q^k_{PAa}, dom^k_{ctrl}, f^k_{path} \rangle :=$ top $Stack$ `// If` $k = 0$ `then` $f^k_{path} \equiv \mathbf{true}$
    `// The successors of` q^k_{PAa} `have been visited?.`
    if $q^k_{PAa} \in Visited$ **then**
        pop $Stack$
        continue
    end
    $Visited := Visited \cup \{ q^k_{PAa} \}$
    foreach *extension* $(q^k_{PAa}, t^k_{PAa}, q^{k+1}_{PAa})$ *of* $\langle q^k_{PAa}, dom^k_{ctrl}, f^k_{path} \rangle$ **do**
        $Q_{PAa} := Q_{PAa} \cup \{ q^{k+1}_{PAa} \}$
        $T_{PAa} := T_{PAa} \cup \{ t^k_{PAa} \}$
        $R_{PAa} := R_{PAa} \cup \{ (q^k_{PAa}, t^k_{PAa}, q^{k+1}_{PAa}) \}$
        `// Is` $(q^{k+1}_{PAa}, q^{k+1}_{ctrl})$ `visited on the current path?`
        `// The test holds vacuously for` final_{PAa}
        if $\neg(dom^{k+1}_{ctrl} = \emptyset \wedge \exists\, q^l_{PAa} \in Stack \cap Visited.\ q^{k+1}_{PAa} \subseteq q^l_{PAa} \wedge dom^l_{ctrl} = \emptyset)$ **then**
            $f^k_{step} := [\![PclStep]\!]_{bool} \wedge \left[\!\!\left[[\![t^k_{PAa}]\!]_{=_{PA}} \right]\!\!\right]_{bool}$
            $dom^{k+1}_{ctrl} := ControlEnvDom\,(dom^k_{ctrl}, fg^k_c, e^k_{ctrl})$
            `// Perform the substitution in Equation 6.65`
            $\widetilde{f^k_{step}} :=$ *update*$(f^k_{step}, dom^{k+1}_{ctrl})$
            push $Stack$, $\langle q^{k+1}_{PAa}, dom^{k+1}_{ctrl}, f^k_{path} \wedge \widetilde{f^k_{step}} \rangle$
        end
    end
end

Algorithm 6.3: Abstraction Algorithm

lences of concrete parcel automaton paths. The algorithm pushes a new abstract state q_{PAa}^{k+1} onto the stack at line 27 if $f_{path}^{k+1} = f_{path}^{k} \wedge \widetilde{f_{step}^{k}}$ is satisfiable and

$$\neg(dom_{ctrl}^{k+1} = \emptyset \wedge \exists\, q_{PAa}^{l} \in Stack \cap Visited.\ q_{PAa}^{k+1} \subseteq q_{PAa}^{l} \wedge dom_{ctrl}^{l} = \emptyset) \tag{6.73}$$

We state an equivalent condition in terms of the corresponding concrete paths $\pi_{PA\,c}^{k+1}$ that satisfy $f_{path}^{k+1} = f_{path}^{k} \wedge \widetilde{f_{step}^{k}}$ that implies that the negation of Equation 6.73 holds. We show that the number of concrete paths that do not satisfy this condition is finite. Therefore, Equation 6.73 is true only for a finite number of cases.

We recall the notion of a variable's driver introduced on page 142. The driver of a variable v_2 at step k is a variable v_1 at step n such that variable v_1 at step n propagates through copying into the value of v_2 at step k.

We say a concrete path $\pi_{PA\,c}^{k}$ is *visited* if during the execution of the algorithm the stack contains a tuple $\langle q_{PAa}^{k}, dom_{ctrl}^{k}, f_{path}^{k} \rangle$ such that $\pi_{PA\,c}^{k} \models f_{path}^{k}$. The stack grows at line 10 or at line 27.

6.5.1 Definition (Explored Path)**.**

$$ExploredPath\ \pi_{PA\,c}^{k} \equiv$$
$$\neg \left(\begin{array}{l} \exists\, n_0 \leq k.\ \exists\, n_0 + 1 \leq n_1 < k. \\ \quad \forall\, i \in \{\, n_0, \ldots, n_1 \,\}.\ \mathit{dom}\, q_{PAc}^{i+1} \left[\mathit{NextRegPcl}/_{\mathit{RegPcl}} \right] \subseteq StateFanOut^{i} \\ \qquad \wedge \\ \qquad\quad \forall\, v \in \mathit{dom}\, q_{PAc}^{n_1+1}.\ driver\,(v, n_1 + 1) = driver\,(v, n_0 + 1) \end{array} \right)$$

We define the dual predicate $NotExploredPath$ that characterizes the paths that are not explored:

$$NotExploredPath\ \pi_{PA\,c}^{k+1} \equiv$$
$$\left(\begin{array}{l} \exists\, n_0 \leq k.\ \exists\, n_0 + 1 \leq n_1 < k. \\ \quad \forall\, i \in \{\, n_0, \ldots, n_1 \,\}.\ \mathit{dom}\, q_{PAc}^{i+1} \left[\mathit{NextRegPcl}/_{\mathit{RegPcl}} \right] \subseteq StateFanOut^{i} \\ \qquad \wedge \\ \qquad\quad \forall\, v \in \mathit{dom}\, q_{PAc}^{n_1+1}.\ driver\,(v, n_1 + 1) = driver\,(v, n_0 + 1) \end{array} \right)$$

6.5.2 Lemma (Termination Of Abstraction Algorithm)**.** If the runs of the concrete parcel automaton $pa_{c\,1}$ are terminating then Algorithm 6.3 terminates.

Proof. **Part 1** We show that if a path satisfies the $NotExploredPath$ predicate then it is not visited.

To show a path is not visited it suffices to show it has a prefix that is not visited. Therefore, assume

$\pi^{k+1}_{PA\,c}$ is minimal to satisfy $NotExploredPath$, i.e. has no strict prefixes that satisfy it. If

$$\left(\begin{array}{l} \exists\, n_0 \leq k.\ \exists\, n_0 + 1 \leq n_1 < k. \\ \quad \forall\, i \in \{\, n_0,\, \ldots,\, n_1 \,\}.\ \mathit{dom}\, q^{i+1}_{PAc} \left[\mathit{NextRegPcl}/_{\mathit{RegPcl}}\right] \subseteq StateFanOut^i \\ \qquad \wedge \\ \qquad\quad \forall\, v \in \mathit{dom}\, q^{n_1+1}_{PAc}.\ driver\ (v, n_1 + 1) = driver\ (v, n_0 + 1) \end{array}\right)$$

then since $\pi^{n_1+1}_{PA\,a}$ is by construction equivalent to $\pi^{n_1+1}_{PA\,c}$ we have that $\pi^{n_1+1}_{PA\,a}$ satisfies the similar property:

$$\left(\begin{array}{l} \exists\, n_0 \leq k.\ \exists\, n_0 + 1 \leq n_1 < k. \\ \quad \forall\, i \in \{\, n_0,\, \ldots,\, n_1 \,\}.\ \mathit{dom}\, q^{i+1}_{PAa} \left[\mathit{NextRegPcl}/_{\mathit{RegPcl}}\right] \subseteq StateFanOut^i \\ \qquad \wedge \\ \qquad\quad \forall\, v \in \mathit{dom}\, q^{n_1+1}_{PAa}.\ driver\ (v, n_1 + 1) = driver\ (v, n_0 + 1) \end{array}\right) \tag{6.74}$$

Given the implication

$$(\forall\, v \in \mathit{dom}\, q^{n_1+1}_{PAa}.\ driver\ (v, n_1 + 1) = driver\ (v, n_0)) \Longrightarrow q^{n_1+1}_{PAa} \subseteq q^{n_0}_{PAa}$$

we obtain that $q^{n_1+1}_{PAa} \subseteq q^{n_0}_{PAa}$. And therefore the path $\pi^{n_1+1}_{PA\,a}$ is not visited and since $\pi^{n_1+1}_{PA\,a}$ is a prefix of $\pi^{k+1}_{PA\,a}$, the latter is not visited either.

Part 2 We show that the set of paths that do not satisfy $NotExploredPath$ is finite:

$$\begin{array}{rcl} ExploredPaths & \equiv & \{\, \pi^{k}_{PA\,c} \mid k \in \mathbf{N} \wedge ExploredPath\ \pi^{k}_{PA\,c} \,\} \\ |ExploredPaths| & < & \infty \end{array} \tag{6.75}$$

The proof of Equation 6.75 is based on showing the claim below, which implies that $ExploredPaths$ contains only paths of length up to a constant k_0 and therefore it is finite.

$$\begin{array}{l} \exists\, k_0.\ \forall\, k \geq k_0. \\ \quad \forall\, \pi^{k+1}_{PA\,c}. \\ \qquad NotExploredPath\ \pi^{k+1}_{PA\,c} \end{array} \tag{6.76}$$

To prove Equation 6.76 we observe that a path $\pi^{k+1}_{PA\,c}$ may contain at most $|Q_{PAc\,1}|$ transitions that do not exclusively copy the parcel's state into the next state:

$$\begin{array}{rcl} NonCopyTrans & \equiv & \{\, i \mid (\mathit{dom}\, q^{i+1}_{PAc}) \left[\mathit{NextRegPcl}/_{\mathit{RegPcl}}\right] \not\subseteq StateFanOut^i \,\} \\ |NonCopyTrans| & \leq & |Q_{PAc\,1}| \end{array} \tag{6.77}$$

Equation 6.77 is proven by contradiction. If it does not hold we can construct a run σ_{PAc} that does not terminate. If $|NonCopyTrans| > |Q_{PAc\,1}|$ then there must be two indices $i_1 < i_2$ in $NonCopyTrans$ such that $q^{i_1}_{PAc} = q^{i_2}_{PAc}$. We then construct the non-terminating run σ_{PAc} as follows:

$$\sigma_{PAc} \equiv \pi^{i_1}_{PA\,c} \xrightarrow{t^{i_1}_{PAc}} \cdots \xrightarrow{t^{i_2-1}_{PAc}} q^{i_1}_{PAc} \xrightarrow{t^{i_1}_{PAc}} \cdots \xrightarrow{t^{i_2-1}_{PAc}} q^{i_1}_{PAc} \cdots$$

We now return to Equation 6.76. We need to find k_0 large enough so that $\pi^{k+1}_{PA\,c}$ admits a large enough subsequence of transitions that only copy the parcel's state:

$$\begin{array}{l} \exists\, n < m \leq k. \\ \quad \forall\, i \in \{\, n, \ldots, m \,\}.\ (\mathit{dom}\ q^{i+1}_{PAc})\ \left[\mathit{NextRegPcl}/_{\mathit{RegPcl}}\right] \subseteq StateFanOut^i \end{array} \tag{6.78}$$

For $i \in \{\, n+1, \ldots, m \,\}$ we define the function

$$driver\ i = \{\, (v, driver\ (v, i)) \mid v \in \mathit{dom}\ q^i_{PAc} \,\}$$

Since for all states at indices between $n+1$ and m the parcel registers take their value from parcel values at step $n+1$, the number of different values that $driver$ can take is bounded by

$$2^{|Q_{PAc\,1}| \times |Q_{PAc\,1}| \times |\mathit{dom}\ q^{n+1}_{PAc}|} \leq 2^{|Q_{PAc\,1}| \times |Q_{PAc\,1}| \times |\mathit{RegPcl}|}$$

Therefore when $m-n-1$ is large enough there will exist n_0 and n_1 such that $driver\ n_0 = driver\ n_1$ which implies

$$\forall\, v \in \mathit{dom}\ q^{n_1+1}_{PAc}.\ driver\ (v, n_1+1) = driver\ (v, n_0)$$

Denote by t the number of non-copying transitions in $\pi^{k+1}_{PA\,c}$ and by s the largest subsequence of copying transitions in $\pi^{k+1}_{PA\,c}$. There are at most $t+1$ copying subsequences in $\pi^{k+1}_{PA\,c}$ because copying subsequences are separated by at least one non-copying transition. We therefore obtain:

$$\begin{array}{rcl} (t+1) \times s + t & \geq & k \\ s & \geq & \dfrac{k-t}{t+1} \\ s & \geq & \dfrac{k - |Q_{PAc\,1}|}{|Q_{PAc\,1}| + 1} \text{ (since } t \leq Q_{PAc\,1}) \end{array}$$

The longest subsequence is at least $\dfrac{k - |Q_{PAc\,1}|}{|Q_{PAc\,1}| + 1}$. Therefore, we choose k_0 so that

$$\frac{k_0 - |Q_{PAc\,1}|}{|Q_{PAc\,1}| + 1} > 2^{|Q_{PAc\,1}| \times |Q_{PAc\,1}| \times |\mathit{RegPcl}|}$$

□

The following theorem states correctness of the abstraction algorithm.

6.5.3 Theorem (Path Abstraction)**.** If the runs of the concrete parcel automaton $pa_{c\,1}$ are terminating then Algorithm 6.3 performs path abstraction of the concrete parcel automaton $pa_{c\,1}$.

Proof. We use Lemma 6.5.2 that states that the algorithm visits a finite number of concrete paths, which implies that if a state $q^k_{PA\,a}$ is on the stack then there is a point in the execution of the algorithm when $q^k_{PA\,a}$ is popped off the stack and extended by Algorithm 6.1.

We prove the following two claims by induction.

(i) If $ExploredPath\ \pi^k_{PA\,c}$ then the algorithm creates the abstract state $q^k_{PA\,a}$ that is reachable along a path $\pi^k_{PA\,a} =_{PA} \pi^k_{PA\,c}$ and pushes onto the stack the tuple $\langle q^k_{PA\,a}, dom^k_{ctrl}, f^k_{path}\rangle$ such that $\pi^k_{PA\,c} \models f^k_{path}$.

(ii) For any $\pi^k_{PA\,c}$ the abstraction algorithm constructs an abstract state $q^k_{PA\,a}$ which is reachable along a path $\pi^k_{PA\,a}$ equivalent to $\pi^k_{PA\,c}$.

Base Case $k = 0$. The concrete path $\pi^0_{PA\,c}$ consists of a single initial state $q^0_{PA\,c}$. At lines 4–11 the algorithm pushes onto the stack the state $q^0_{PA\,a} =_{PA} q^0_{PA\,c}$ and therefore, $\pi^0_{PA\,c} =_{PA} \pi^0_{PA\,a}$.

Inductive Case We assume that the two claims are true for $i \leq k$ and prove it for $k+1$.

Claim (i) We need to show that if $ExploredPath\ \pi^{k+1}_{PA\,c}$ then the algorithm creates the abstract state $q^{k+1}_{PA\,a}$ that is reachable along a path $\pi^{k+1}_{PA\,a} =_{PA} \pi^{k+1}_{PA\,c}$ and pushes onto the stack the tuple $\langle q^{k+1}_{PA\,a}, dom^{k+1}_{ctrl}, f^{k+1}_{path}\rangle$ such that $\pi^{k+1}_{PA\,c} \models f^{k+1}_{path}$.

Since $ExploredPath\ \pi^{k+1}_{PA\,c}$ holds, its prefix $\pi^k_{PA\,c}$ also satisfies the predicate $ExploredPath$. By induction, the algorithm puts on the stack the tuple $\langle q^k_{PA\,a}, dom^k_{ctrl}, f^{k-1}_{path} \wedge \widetilde{f^{k-1}_{step}}\rangle$ such that $\pi^k_{PA\,c} \models f^{k-1}_{path} \wedge \widetilde{f^{k-1}_{step}}$. And further, $q^k_{PA\,a}$ is reachable via a path $\pi^k_{PA\,a} =_{PA} \pi^k_{PA\,c}$.

Since only a finite number of paths are visited, the tuple $\langle q^k_{PA\,a}, dom^k_{ctrl}, f^k_{path}\rangle$ is eventually popped off the stack at line 13. The state $q^k_{PA\,a}$ is then extended by abstract steps corresponding to the equivalence classes of concrete paths of length $k+1$ that satisfy the formula $f^k_{path} \wedge [PclStep]_{bool}(\widetilde{V_{step}}^k)$. Since $\pi^k_{PA\,c} \models f^k_{path}$, $\pi^{k+1}_{PA\,c} \models f^k_{path} \wedge [PclStep]_{bool}(\widetilde{V_{step}}^k)$. The path extension algorithm therefore constructs an abstract step $t^k_{PA\,a}$ that is equivalent to $t^k_{PA\,c}$ and an abstract state $q^{k+1}_{PA\,a}$ reachable by a path equivalent to $\pi^{k+1}_{PA\,c}$. The abstraction algorithm then performs the check at line 27 which returns false since $ExploredPath\ \pi^{k+1}_{PA\,c}$ holds. The tuple $\langle q^{k+1}_{PA\,a}, dom^{k+1}_{ctrl}, f^k_{path} \wedge \widetilde{f^k_{step}}\rangle$ such that $\pi^{k+1}_{PA\,c} \models f^k_{path} \wedge \widetilde{f^k_{step}}$ is then pushed onto the stack.

Claim (ii) If $\pi^{k+1}_{PA\,c}$ satisfies $ExploredPath$ then this case reduces to the previous one.

Consider the maximal prefix $\pi^{n}_{PA\,c}$ of $\pi^{k+1}_{PA\,c}$ such that $ExploredPath\ \pi^{n}_{PA\,c}$ holds. By induction the algorithm visits the path $\pi^{n_1}_{PA\,c}$ and extends the equivalent abstract path $\pi^{n_1}_{PA\,a}$ so that

$$\pi^{n_1+1}_{PA\,c} =_{PA} \pi^{n_1+1}_{PA\,a}$$

However, the path $\pi^{n_1+1}_{PA\,c}$ is not visited because

$$\left(\begin{array}{l} \exists\, n_0 \leq k.\ \exists\, n_0 + 1 \leq n_1 < k. \\ \quad \forall\, i \in \{\, n_0, \ldots, n_1 \,\}.\ \mathit{dom}\, q^{i+1}_{PAc} \left[\mathit{NextRegPcl}/_{\mathit{RegPcl}}\right] \subseteq StateFanOut^i \\ \qquad \wedge \\ \qquad\quad \forall\, v \in \mathit{dom}\, q^{n_1+1}_{PAc}.\ driver\,(v, n_1+1) = driver\,(v, n_0+1) \end{array}\right) \tag{6.79}$$

Equation 6.79 implies that $q^{n_1+1}_{PAc} \subseteq q^{n_0+1}_{PAc}$, therefore

$$\pi_{PAc\,1} = \pi^{n_0+1}_{PA\,c}\ t^{n_1+1}_{PAc} \cdots t^{k}_{PAc}\ q^{k+1}_{PAc}$$

is a path in $pa_{c\,1}$ of length less equal to k. By induction there exists an equivalent abstract path $\pi_{PAa\,1} =_{PA} \pi_{PAc\,1}$. Since the equivalence class of a concrete path is visited only once, the path $\pi_{PAa\,1}$ continues the path $\pi^{n_0+1}_{PA\,a}$.

$$\pi_{PAa\,1} = \pi^{n_0+1}_{PA\,a}\ t^{n_1+1}_{PAa} \cdots t^{k}_{PAa}\ q^{k+1}_{PAa}$$

We modify the path $\pi_{PAa\,1}$ by splicing in the path segment

$$q^{n_0+1}_{PAa}\ t^{n_0+1}_{PAa} \cdots q^{n_1}_{PAa}\ t^{n_1}_{PAa}\ q^{n_1+1}_{PAa}$$

The resulting path $\pi_{PAa\,2}$ is equivalent to $\pi^{k+1}_{PA\,c}$.

$$\pi_{PAa\,2} = \pi^{n_0+1}_{PA\,a}\ t^{n_0+1}_{PAa} \cdots q^{n_1}_{PAa}\ t^{n_1}_{PAa}\ q^{n_1+1}_{PAa}\ t^{n_1+1}_{PAa} \cdots t^{k}_{PAa}\ q^{k+1}_{PAa}$$

□

Because abstract states could be reached on different paths in the DFS algorithm, due to either back edges or cross edges, it is not generally the case that

$$\mathcal{L}(pa_{a}) \subseteq \mathcal{L}(\overline{pa}_{c\,1})$$

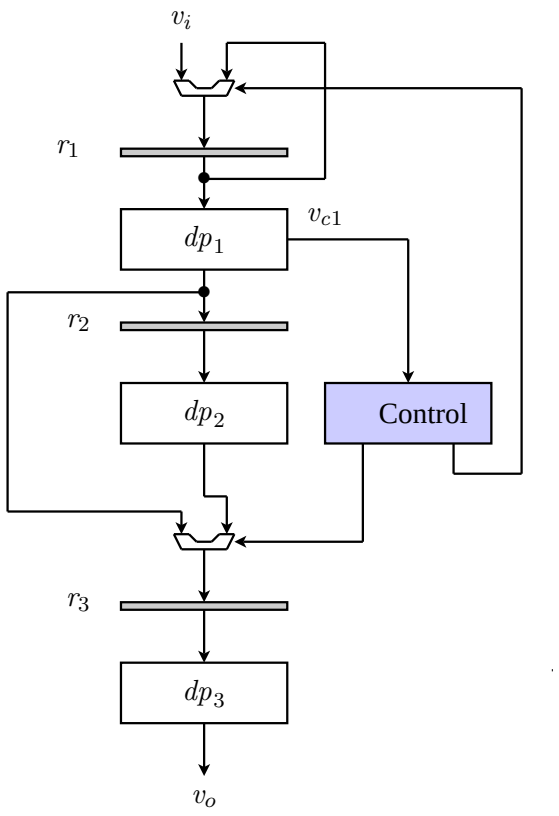


Figure 6.4: Pipeline model with stall and exclusive paths.

We use the example in Figure 6.4 to illustrate the result produced by the abstraction algorithm. There are two possible paths through the pipeline:

$$v_i \longrightarrow r_1 \longrightarrow r_2 \longrightarrow r_3 \longrightarrow v_o$$
$$v_i \longrightarrow r_1 \longrightarrow r_3 \longrightarrow v_o$$

When the parcel in r_1 produces output $v_{c1} = 0$, it transfers to r_2. When $v_{c1} = 1$ it should transfer to r_3. If both the parcel in r_1 and r_2 need to transfer to r_3, the one in r_2 is given priority. This means the parcel in r_1 stalls.

The concrete parcel automaton $pa_{c\,1}$ is succinctly described in Figure 6.5. The figure represents the two types of paths that are possible through the pipeline and uses symbolic values. Transition labels that such that $e_{ctrlc} = \emptyset$ are not shown. The result of Algorithm 6.3 is shown in Figure 6.6. The two concrete paths $\emptyset \longrightarrow \{\, r_1 = a_0 \,\}$ and $\emptyset \longrightarrow \{\, r_1 = b_0 \,\}$ are indistinguishable during abstraction and are therefore represented by the same abstract path $\emptyset \longrightarrow \{\, r_1 = \alpha_0 \,\}$. The abstract value α_0 stands for both a_0 and b_0. When the state $\{\, r_1 = \alpha_0 \,\}$ is expanded, there are two abstract transitions

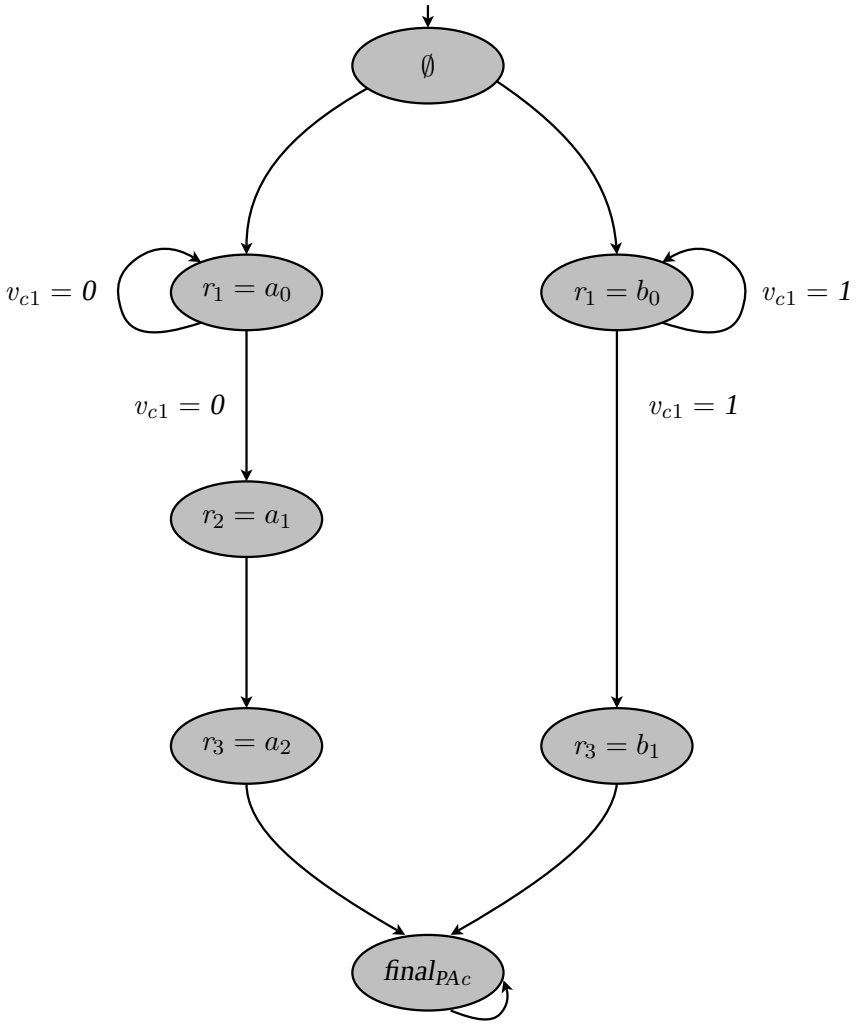


Figure 6.5: Partial representation of $pa_{c\,1}$ using symbolic values.

satisfied by a_0 and another two satisfied by b_0. Because of the self loops $\{\, r_1 = \alpha_0 \,\} \overset{v_{c1} = 0}{\longrightarrow} \{\, r_1 = \alpha_0 \,\}$ and $\{\, r_1 = \alpha_0 \,\} \overset{v_{c1} = 1}{\longrightarrow} \{\, r_1 = \alpha_0 \,\}$ the abstract automaton can represent paths that are a mix of distinct paths of the concrete automaton:

$$\emptyset \longrightarrow \{\, r_1 = \alpha_0 \,\} \overset{v_{c1} = 0}{\longrightarrow} \{\, r_1 = \alpha_0 \,\} \overset{v_{c1} = 1}{\longrightarrow} \{\, r_3 = \beta_0 \,\} \longrightarrow final_{PAa}$$

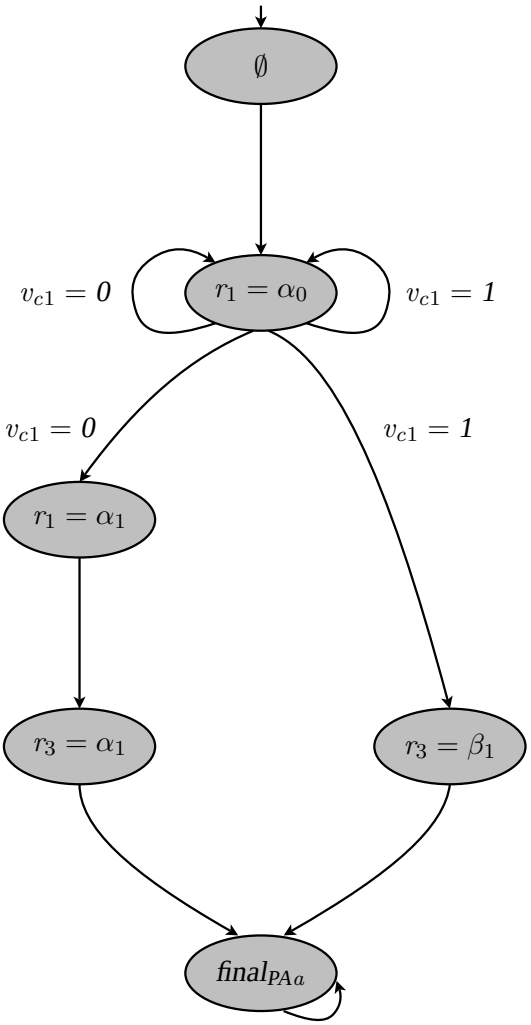


Figure 6.6: Partial representation of pa_a constructed by Algorithm 6.3.

In the rest of the section we generalize the idea behind Algorithm 6.3 so that $\mathcal{L}(pa_a) = \mathcal{L}(pa_{c\,1})$. The abstract parcel automaton contains paths that are not possible in the concrete one due to the fact that constraints corresponding to paths that have a common prefix are solved independently. The constraint the algorithm uses when expanding an abstract state corresponds to the conjunction of the step formulas that represent the path currently explored. The natural generalization is to replace path formulas by a conjunction of formulas representing all the abstract transitions discovered so far by the DFS procedure.

We recall the representation of parcel steps in propositional logic introduced in Section 6.3. The set of variables that encode the parcel step is defined as follows:

$$\widetilde{V_{step}}^{k} = \widetilde{\mathit{RegPcl}}^{k} \uplus \widetilde{\mathit{CombPcl}}^{k} \uplus \widetilde{\mathit{NextRegPcl}}^{k+1} \uplus \widetilde{ControlVars}^{k} \uplus \widetilde{V_{Dps}}^{k} \uplus \widetilde{Parcel}^{k} \uplus \widetilde{V_{fanOut}}^{k}$$

and the formula that encodes a parcel step from a state $q^k_{\mathit{PA}c}$ is given by $[\![PclStep]\!]_{bool}\,(\widetilde{V_{step}}^{k})$. The equivalence class of a parcel step was represented by the formula below:

$$f^k_{step} \equiv [\![PclStep]\!]_{bool}\,(\widetilde{V_{step}}^{k}) \wedge \left[\!\!\left[\left[\!\!\left[t^k_{\mathit{PA}c} \right]\!\!\right]_{=_{\mathit{PA}}} \right]\!\!\right]_{bool} (\widetilde{V_{step}}^{k})$$

According to Definition 6.3.4 we write

$$(q^k_{\mathit{PA}c}, t^k_{\mathit{PA}c}, q^{k+1}_{\mathit{PA}c}) \models f^k_{step}(\widetilde{V_{step}}^{k})$$

if there exists an environment $e^k_{step\,c} \in \mathit{Env}(\widetilde{V_{step}}^{k})$ satisfying Equation 6.52 up to Equation 6.57 and

$$e^k_{step\,c} \models f^k_{step}(\widetilde{V_{step}}^{k})$$

Given a propositional formula f, v a variable and $V = \{\, v_1, \ldots, v_n \,\}$ a set of variables, we use the following standard notation:

$$\begin{aligned} (\exists\, v)\, f &\equiv \bigvee_{d \in \mathit{Ty}(v)} f\left[{}^{d}/_{v}\right] \\ (\exists\, V)\, v &\equiv (\exists\, v_1)\,(\cdots (\exists\, v_n)\, f) \end{aligned}$$

It follows that the existence of a step equivalent to $(q^k_{\mathit{PA}c}, t^k_{\mathit{PA}c}, q^{k+1}_{\mathit{PA}c})$ from state $q^k_{\mathit{PA}c}$ is equivalent to the satisfiability of the following propositional formula:

$$\left[\!\!\left[\mathit{HasStep}(\left[\!\!\left[t^k_{\mathit{PA}c} \right]\!\!\right]_{=_{\mathit{PA}}}) \right]\!\!\right]_{bool} (\widetilde{\mathit{RegPcl}}^{k}) \equiv (\exists\, \widetilde{V_{step}}^{k} \setminus \widetilde{\mathit{RegPcl}}^{k})\, f^k_{step}$$

The path extension procedure (Algorithm 6.1) uses the satisfiability solver to extract all the concrete

parcel steps

$$\{\,(q^k_{PAc}, t^k_{PAc\,1}, q^k_{PAc\,1}),\ \ldots,\ (q^k_{PAc}, t^k_{PAc\,m}, q^k_{PAc\,m})\,\}$$

that are possible from a concrete state q^k_{PAc} constrained by $Constraint_k$, where $Constraint_k = f^k_{path}$ in the first version of the abstraction algorithm.

We say the non-empty set of steps

$$\{\,(q^k_{PAc}, t^k_{PAc\,i_1}, q^k_{PAc\,i_1}),\ \ldots,\ (q^k_{PAc}, t^k_{PAc\,i_r}, q^k_{PAc\,i_r})\,\}$$

is *exact* if the following formula

$$\left(\begin{array}{c} Constraint_k \\ \wedge \\ \bigwedge\limits_{j\in\{\,i_1,\ldots,i_r\,\}} \Big[\!\Big[HasStep(\,\big[\!\big[t^k_{PAc\,j} \big]\!\big]_{=_{PA}}) \Big]\!\Big]_{bool} (\widetilde{RegPcl}^{\,k}) \\ \wedge \\ \bigwedge\limits_{j\in\{\,1,\ldots,m\,\}\setminus\{\,i_1,\ldots,i_r\,\}} \neg \Big[\!\Big[HasStep(\,\big[\!\big[t^k_{PAc\,j} \big]\!\big]_{=_{PA}}) \Big]\!\Big]_{bool} (\widetilde{RegPcl}^{\,k}) \end{array}\right)$$

is satisfiable.

Note that to represent an exact set of parcel steps in a formula $Constraint_k$ we must have multiple copies of the set of variables $\widetilde{V_{step}}^{\,k}$.

$$\widetilde{V_{step\,j}}^{\,k} \equiv \widetilde{RegPcl}^{\,k} \uplus \widetilde{CombPcl}^{\,k}_j \uplus \widetilde{NextRegPcl}^{\,k+1}_j \uplus \widetilde{ControlVars}^{\,k}_j \uplus \widetilde{V_{Dps\,j}}^{\,k} \uplus \widetilde{Parcel}^{\,k}_j \uplus \widetilde{V_{fanOut\,j}}^{\,k}$$

$$f^k_{step\,j} \equiv [PclStep]_{bool}\,(\widetilde{V_{step\,j}}^{\,k}) \wedge \Big[\!\Big[\big[\!\big[t^k_{PAa\,j} \big]\!\big]_{=_{PA}} \Big]\!\Big]_{bool} (\widetilde{V_{step\,j}}^{\,k})$$

The generalized abstraction algorithm is shown in Algorithm 6.4. The actual abstraction is performed by the recursive procedure *AbstractRec* described in Algorithm 6.5.

The idea in algorithm in Algorithm 6.5 is similar to the one in Algorithm 6.3. The generalized algorithm finds exact sets of parcel steps that continue the abstract state that was popped off the stack. For each such subset it makes a recursive call that passes the updated DFS state of the current instance of *AbstractRec*. The algorithm makes use of a function *CopyPcl* that returns a fresh copy of the DFS digraph (partially constructed abstract parcel automaton). If a recursive call is made the current instance returns since the recursive calls cover all the possible cases.

6.5.4 Proposition. If the runs of the concrete parcel automaton $pa_{c\,1}$ are terminating then Algorithm 6.4 terminates.

Input: $Pipe_c$
Output: $\langle Q_{PAa}, R_{PAa}, T_{PAa}, I_{PAa} \rangle$

1 $\langle Q_{PAa}, R_{PAa}, T_{PAa}, I_{PAa} \rangle := \langle \{ \mathit{final}_{PAa} \}, \emptyset, \emptyset, \emptyset \rangle$

2 **foreach** $p \in InitParcels$ **do**
3 $\quad q^0_{PAa} := \mathit{newAbstractState}(p)$
4 $\quad dom^0_{ctrl} := \emptyset$
5 $\quad I_{PAa\,0} := \{ q^0_{PAa} \}$
6 $\quad Q_{PAa\,0} := \{ q^0_{PAa} \}$
7 $\quad R_{PAa\,0} := \emptyset$
8 $\quad T_{PAa} := \emptyset$
9 $\quad Constraint_0 := \emptyset$
10 $\quad Stack_0 := \{ \langle q^0_{PAa}, dom^0_{ctrl} \rangle \}$
11 $\quad Visited_0 := \emptyset$
12 $\quad \mathit{AbstractRec}\, \langle \langle I_{PAa\,0}, Q_{PAa\,0}, R_{PAa\,0}, T_{PAa\,0} \rangle, Constraint_0, Stack_0, Visited_0 \rangle$
13 **end**

Algorithm 6.4: Abstraction Algorithm II

Proof. The same technique used in Lemma 6.5.2 to prove termination of Algorithm 6.3 is also applicable here. □

6.5.5 Theorem. If the runs of the concrete parcel automaton $pa_{c\,1}$ are terminating then the abstract parcel automaton pa_a has the same language as the concrete parcel automaton $pa_{c\,1}$.

Proof. The abstract parcel automaton consists of the union of the parcel automata returned at lines 25–28 in Algorithm 6.5. These parcel automata can only share the set of states $\{ \emptyset, \mathit{final}_{PAa} \}$. When such a parcel automaton is returned, all the edges of the finite paths that it represents are encoded in the global constraint $Constraint_n$ in the procedure *AbstractRec*. The invariant of the algorithm is that the global constraint is always satisfiable. It therefore follows that for each finite path through pa_a there exists an equivalent one in $pa_{c\,1}$. We can apply Lemma 6.1.4 on page 143 to prove the inclusion $\mathcal{L}(pa_a) \subseteq_{PA} \mathcal{L}(pa_{c\,1})$. The other inclusion holds since the algorithm performs path abstraction. □

It is possible that for the abstract parcel automaton pa_a returned by Algorithm 6.4 language equality between the concrete and abstract pipelines does not hold. This happens when parcel input variables and parcel variables that are assigned **choice** occur in several parcel steps of the abstract parcel automaton. In this case, *Dps* pa_a returns **choice** for a combination of values that does not show up on the edges of the parcel automaton.

To solve this problem, we need to modify the parcel automata returned at lines 25–28. Instead of a parcel automaton with abstract values *AbstractRec* returns one based on a solution to the constraint

Input: $\langle\langle I_{PAa\,n}, Q_{PAa\,n}, R_{PAa\,n}, T_{PAa\,n}\rangle, Constraint_n, Stack_n, Visited_n\rangle$
Output: updates $\langle I_{PAa}, Q_{PAa}, R_{PAa}, T_{PAa}\rangle$

while $Stack_n \neq \emptyset$ **do**
  $\langle q^k_{PAa}, dom^k_{ctrl}\rangle :=$ pop $Stack_n$
  foreach *set of exact steps* $ExactSteps^k_n$ *of* $\langle q^k_{PAa}, Constraint_n\rangle$ **do**
    $\langle I_{PAn+1}, Q_{PAn+1}, R_{PAn+1}, T_{PAa\,n+1}\rangle :=$ *CopyPcl* $\langle I_{PAn}, Q_{PAn}, R_{PAn}, T_{PAa\,n}\rangle\ (n+1)$
    $Stack_{n+1} :=$ *CopyPcl* $Stack_n\ (n+1)$
    $Visited_{n+1} :=$ *CopyPcl* $Visited_n\ (n+1)$
    $ExactSteps^k_{n+1} :=$ *CopyPcl* $ExactSteps^k_n\ (n+1)$
    foreach $(q^k_{PAa}, t^k_{PAa\,j}, q^{k+1}_{PAa\,j}) \in ExactSteps^k_{n+1}$ **do**
      $Q_{PAa\,n+1} := Q_{PAa\,n+1} \cup \{\, q^{k+1}_{PAa\,j} \,\}$
      $T_{PAa\,n+1} := T_{PAa\,n+1} \cup \{\, t^k_{PAa\,j} \,\}$
      $R_{PAa\,n+1} := R_{PAa\,n+1} \cup \{\, (q^k_{PAa}, t^k_{PAa\,j}, q^{k+1}_{PAa\,j}) \,\}$
      $f^k_{step\,j} := [\![PclStep]\!]_{bool}\,(\widetilde{V_{step\,j}}^{\,k}) \wedge \left[\!\!\left[\left[\!\!\left[t^k_{PAa\,j} \right]\!\!\right]_{=_{PA}} \right]\!\!\right]_{bool}$
      $dom^{k+1}_{ctrlj} := ControlEnvDom\ (dom^k_{ctrl}, fg^k_j, e^k_{ctrl})$
      // Perform the substitution in Equation 6.65
      $\widetilde{f^k_{step\,j}} :=$ *update*$(f^k_{step\,j}, dom^{k+1}_{ctrlj})$
      $Constraint_{n+1} := Constraint_n \wedge \widetilde{f^k_{step\,j}}$
      // Is $q^{k+1}_{PAa\,j}$ visited on the current path?
      // The test holds vacuously for $final_{PAa}$
      if $\neg(dom^{k+1}_{ctrlj} = \emptyset \wedge \exists\, q^l_{PAa} \in Stack_{n+1} \cap Visited_{n+1}.\ q^{k+1}_{PAa\,j} \subseteq q^l_{PAa} \wedge dom^l_{ctrl} = \emptyset)$ **then**
        $Visited_{n+1} := Visited_{n+1} \cup \{\, q^{k+1}_{PAa\,j} \,\}$
        push $Stack_{n+1}, \langle q^{k+1}_{PAa\,j}, dom^{k+1}_{ctrlj}\rangle$
      end
    end
    // Recursive call for the current set of successors.
    AbstractRec $\langle\langle Q_{PAa\,n+1}, R_{PAa\,n+1}, T_{PAa\,n+1}\rangle, Constraint_{n+1}, Stack_{n+1}, Visited_{n+1}\rangle$
  end
  // The DFS algorithm was finished by the recursive calls.
  return
end
// There were no recursive calls
$I_{PAa} := Q_{PAa} \cup I_{PAa\,n}$
$Q_{PAa} := Q_{PAa} \cup Q_{PAa\,n}$
$R_{PAa} := R_{PAa} \cup R_{PAa\,n}$
$T_{PAa} := T_{PAa} \cup T_{PAa\,n}$

Algorithm 6.5: Recursive Abstraction Procedure *AbstractRec*.

$Constraint_n$. We then use a simulator to find out the values produced by datapaths under parcel input combinations that are not known from the parcel steps.

6.6 Case Studies

This section describes our implementation results in the Bluenose II design and verification tool. Our abstraction algorithm was tested with several designs: the *DiffAddMult* arithmetic pipeline, an edge detector and a two-wide superscalar OpenRISC microprocessor.

6.6.1 Design For Verification Using *PipeNet*

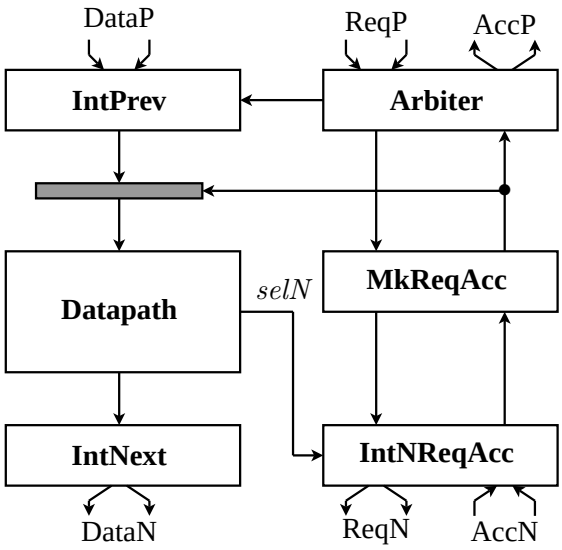


Figure 6.7: Pipeline stage template.

Our case studies were designed using a library of reusable design components called *PipeNet* [Higgins and Aagaard, 2005]. The main building block that *PipeNet* uses is the pipeline stage. With *PipeNet*, the design consists of a collection of interconnected blocks that represent the pipeline circuit and associated memories, register files and other design components such as hazard detection units. Within a stage, *PipeNet* provides a clean separation between datapath and control. The structure of a stage is described in Figure 6.7. The parcel that enters the stage is selected by the mux *InterfacePrev* and is stored in the stage register. When it eventually exits, its value is copied through the demux *InterfaceNext*.

Parcel flow through the stage and through the pipeline is coordinated using a request-accept protocol. Parcels move from one stage to another by generating requests which, if granted an accept, lead to the parcel transferring to one of the requested stages. The request-accept protocol has a distributed implementation. Requests and accepts propagate through the pipeline, without the need of global control.

Inside a stage, the request accept protocol is implemented by the three remaining blocks: *Arbiter*, *MakeReqAcc*, and *InterfaceNextReqAcc*. The role of the arbiter is to prioritize the incoming requests and drive the select signal of the *InterfacePrev* mux. The *MakeReqAcc* block maintains the occupation status of the stage and calculates whether the stage will be able to accept the request of another based on whether its own request has been granted. The request that the parcel in the current stage makes, originates as a control output of the stage datapath, which by convention is called $selN$. *InterfaceNextReqAcc* acts as the relay for the request-accept protocol with the downstream stages.

The concept of pipeline models presented in Chapter 3 is a natural generalization of *PipeNet*. The *PipeNet* template provides clear boundaries between datapath and control. Between stages, parcel values are only copied through muxes or demuxes, which corresponds to the assignment of if-then-else parcel expressions in the pipeline model. Among the generalizations brought by the pipeline model are the dissolution of stage boundaries and the hiding of the request-accept protocol. The stage datapaths are replaced by combinational datapath modules with multiple input and output parcel variables. Datapaths are allowed to consume both primary parcel input variables and parcel registers. The protocol used by *PipeNet* to synchronize multicycle datapaths has also been hidden away.

Stage datapaths can take multiple cycles to compute their result. During this time the pipeline stage is busy, and it will not generate or accept requests and the enable signal of the stage register is not asserted. Multicyle datapaths use a simple protocol to start their computation and to signal when the result is available. An input control signal of the datapath is used to start the computation, and respectively, an output signal of the datapath is asserted in the cycle when the computation is done. In the pipeline model, multicycle datapaths are represented by combinational datapaths that take as additional input and output parameters the registers corresponding to the internal state of the multicyle datapath.

In a *PipeNet* design, the parcel map is defined using the pipeline stages or primary input variables. For combinational datapaths, parcels are singletons that consist of a stage register. For sequential datapaths, the parcel consists of both the stage register and the additional parcel registers of the multicycle datapath. The proof obligations for the parcel map (Section 5.1) are simplified by the fact that parcels are singletons when they transfer into a stage. The state of multicycle datapaths is reset when a new parcel transfers into the stage and therefore, it does not persist after a multicycle computation ends.

6.6.2 Implementation

The abstraction algorithm is implemented in Bluenose II [Chan et al., 2007], a tool for the design and verification of pipelined circuits. The high-level description of a pipeline design is input into the tool using block diagrams in a visual editor based on the Eclipse Graphical Modeling Framework or textually using the Groovy Builder syntax [Koenig et al., 2007].

```
PipeStage(name : 'sub_stage', next : ['neg_stage', 'add_stage', 'mult_stage']) {
  BlkInterface() {
    Parcel(name : 'pclIn', vhdlType : 'pcl_sub_ty', direction : 'IN')
    Parcel(name : 'pclOut', vhdlType : 'pcl_neg_ty', direction : 'OUT')
  }

  Datapath(name : 'sub', vhdlId : 'sub', implFiles : ['sub.vhd']) {
    BlkInterface() {
      Parcel('i_data')
      Parcel('o_data')
      SelN('reqN')
    }
  }
}
```

Figure 6.8: Combinational stage.

We recall the *DiffAddMult* example first introduced in Chapter 3. Figure 6.8 and Figure 6.9 illustrate how a combinational and, respectively, a sequential stage are represented in Bluenose II. The user describes the parcel ports of the stage and provides annotations for the datapath. The datapaths have standalone user provided VHDL implementations which are referenced from model file. Multicycle datapaths have additional annotations describing the start and finish control ports.

Bluenose II generates hierarchical VHDL for the pipeline model and uses the Mentor Graphics Precision RTL synthesis tool to create a hierarchical gate-level netlist of the design. At this stage, the netlist contains black boxes that in a design flow are implemented using the target FPGA technology. We have reverse engineered using trial and error close to twenty such black box operators for which we have provided generic gate-level VHDL implementations. Further rounds of synthesis are performed to rewrite recursively all the black box operators with gatelevel implementations. A translation to the NuSMV model checker [Cimatti et al., 2002] is made available by providing semantics of the primitive gates in the model checker's language. Identifying the correct semantics of the various types of flip-flops was the most challenging aspect in adding support for NuSMV.

We have implemented the first flavour of path abstraction (Algorithm 6.3) in Bluenose II. Our al-

```
PipeStage(name : 'mult_stage') {
  BlkInterface() {
    Parcel(name : 'pclIn', vhdlType : 'pcl_neg_ty', direction : 'IN')
    Parcel(name : 'pclOut', vhdlType : 'pcl_mult_res_ty', direction : 'OUT')
  }

  SequentialDatapath(name : 'mult', implFiles : ['mult.vhd', 'multiplier.vhd']) {
    BlkInterface() {
      Parcel('i_data')
      Parcel('o_data')
      Clock('i_clk')
      Reset('i_reset')
      Start('i_start')
      Finished('o_finished')
      SelN('reqN')
    }
  }
}
```

Figure 6.9: Sequential Stage

gorithm exploits the $selN$ and $finished$ signals produced by the pipeline datapaths to compute the possible parcel steps for the abstract state at the top of the stack. The $selN$ signal uses a one hot-encoding to represent the stage the parcel needs to go next. The successor states of the parcel state correspond to the stages that are encoded by a *1* in the $selN$ vector. For multicycle stages, when the value of the $finished$ signal is *0*, the next parcel automaton state consists of the current parcel's value and the next datapath state. Once abstraction is done, the tool generates VHDL for abstract datapaths represented by the parcel automaton. Control properties of the resulted pipeline are then verified using a model checker.

The path formula corresponding to a multiply operation in the *DiffAddMult* example is shown in Figure 6.10. The path formula is satisfied by computations of the concrete parcel automaton of form

$$Sub\ ^{selN\xrightarrow{Sub}=\,001}\ Neg\ ^{selN\xrightarrow{Sub}=\,10}\ Mult\ ^{finished\xrightarrow{Mult}=\,0}$$

$$Mult\ ^{finished\xrightarrow{Mult}=\,0}\ Mult\ ^{selN\xrightarrow{Mult}=\,0}_{finished_{Mult}=1}$$

In the path formula the first instance of the $Mult$ datapath is used to put the datapath in the reset state. Corresponding to the internal registers of the datapath, the formula contains constraints that

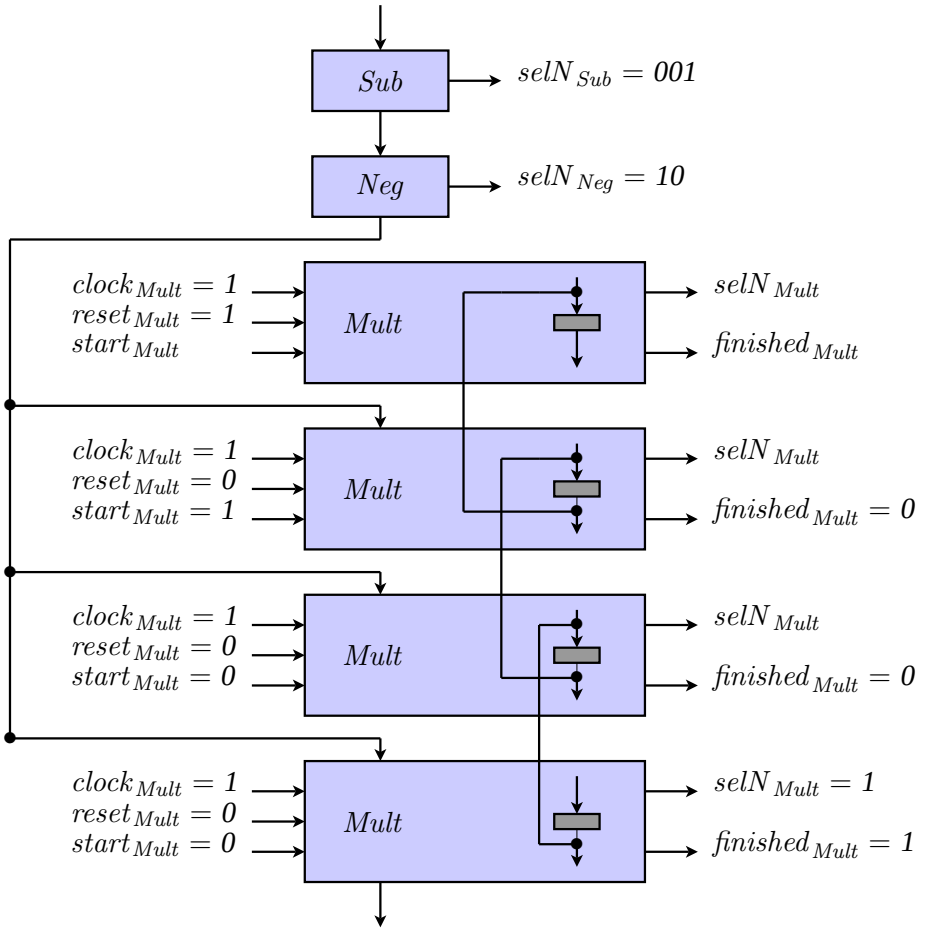


Figure 6.10: Example of path formula for *DiffAddMult*.

ensure updates to the registers in the current cycle propagate into the copy of the registers in the next cycle.

The abstract parcel automaton for *DiffAddMult* is represented in Figure 6.11. Our implementation uses the MiniSat solver [Een and Sorensson]. The datapath of the 32-bit version of *DiffAddMult* has a total of 22881 gates. After abstraction, the total becomes 148. The path problems require 56 SAT problems with a cumulative time of 94 seconds. The maximum memory used is 39.8MB and maximum time is 5 seconds.

Another example that we used illustrates the ease of applying abstraction with Bluenose II to circuits that were not designed with verification in mind. We imported with minimal effort the design of a Kirsch edge detector that was originally created for a course project. The circuit consists of a pipeline that has two multicycle stages. The parcel automaton obtained by abstraction is shown

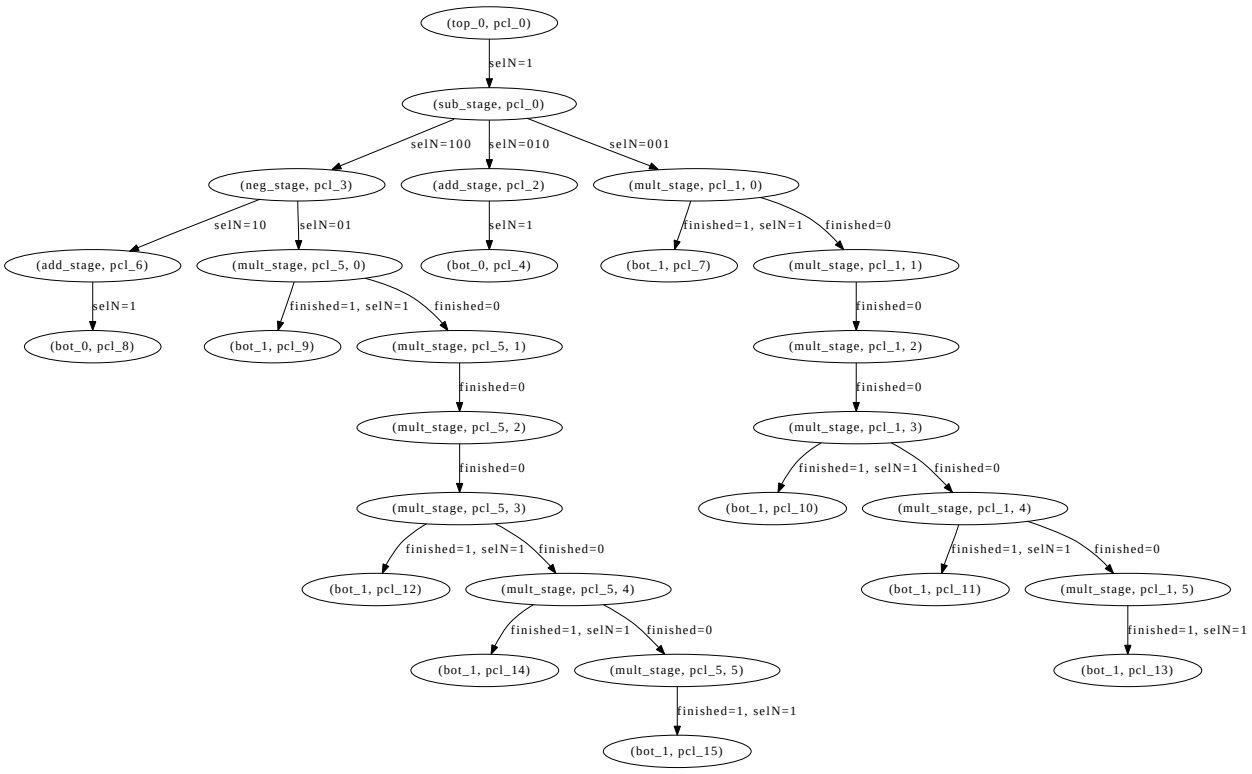


Figure 6.11: *DiffAddMult* abstract parcel automaton.

in Figure 6.12. The concrete datapath had 2200 gates and was reduced to only 144 gates, with a cumulative time of 33 seconds for 32 SAT problems, using less than 3.5MB of memory.

6.6.3 Abstraction Of The OpenRisc Processor

Our OpenRISC processor is a two-wide superscalar pipeline for the ORBIS32 32-bit integer RISC instruction set; it contains a cyclic path, uses bubble squashing, and executes instructions in program order. Our OpenRISC design implements 47 of the 52 instructions — all instructions except those that require operating system support or special-purpose registers. The ORBIS32 instruction set architecture uses a load/store approach and defines a flag condition code register that is used for conditional branch operations.

In Figure 6.13, parcel connections are shown as thick lines and normal signals are shown as thin lines. The three grey parcel connections are secondary paths that can be used to squash bubbles in the event that the primary path is stalled. For example, IF_0 will use the secondary path $\mathrm{IF}_0 \longrightarrow \mathrm{ID}_1$ if ID_0 is stalled and there is a bubble in ID_1. The ALU stage has two groups of multi-cycle instruc-

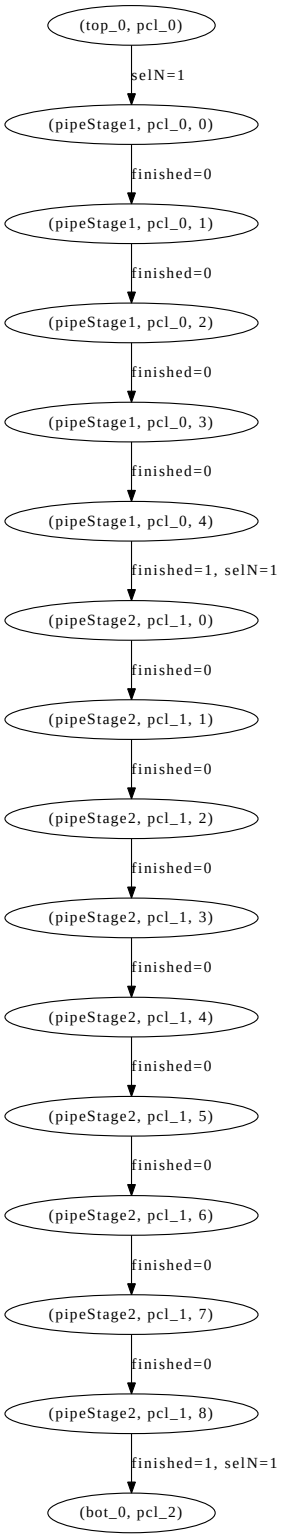


Figure 6.12: Abstract parcel automaton of edge-detector

Instruction Class	Instruction Listing
Arithmetic	add, addc, addi, mul, muli, mulu, sub
Logical	and, andi, or, ori, rori, sll, slli, sra, srai, srl, srli, xor, xori
Control flow	bf, bnf, j, jal, jalr, jr
Flag set	sfeq, sfges, sfgeu, sfgts, sfgtu, sfles, sfleu, sflts, sfltu, sfne
Load/Store	lbs, lbz, lhs, lhz, lws, lwz, sb, sh, sw
Misc	nop, movhi
Not implemented	trap, rfe, mfspr, mtspr, sys

Table 6.2: ORBIS32 Instruction Set

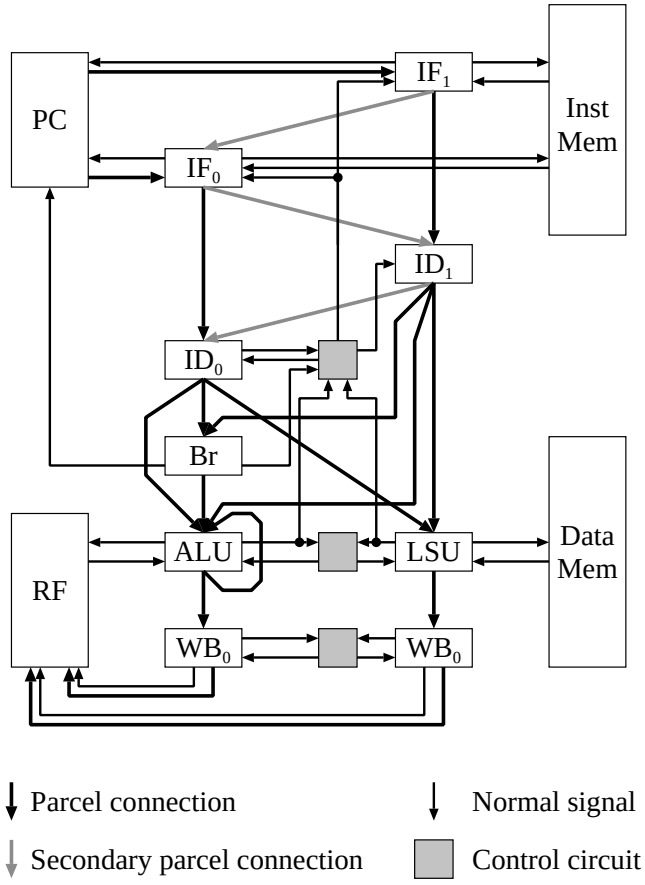


Figure 6.13: OpenRisc pipeline

tions: multiply instructions use a sequential datapath inside the stage and shift/rotate instructions loop through the stage multiple times. We chose these two different methods of doing multi-cycle operations to illustrate that our abstraction supports both. Latencies through the ALU vary from one clock cycle for simple instructions to four clock cycles for multiplications.

Structural reduction [Beer et al., 1994] removes the register file, instruction memory, data memory,

and program counter. The source end of the parcel connections from the program counter to IF_0 and IF_1 become unconnected. Similarly, the target end of the parcel connections from the writeback stages become unconnected. Each unconnected source end is then connected to a top pseudo stage and each unconnected target end is connected to a bottom pseudo stage. The grey control circuits are preserved, because they are connected to control circuits within stages, not to the datapaths. The unconnected source ends on non-parcel data signals to datapaths (e.g. , data outputs from the register file and memory) become non-deterministic inputs to the datapaths. We then apply our abstraction algorithm (Algorithm 6.3) to the pipeline, replace the datapaths in the pipeline with the abstractions, and generate an abstract pipeline.

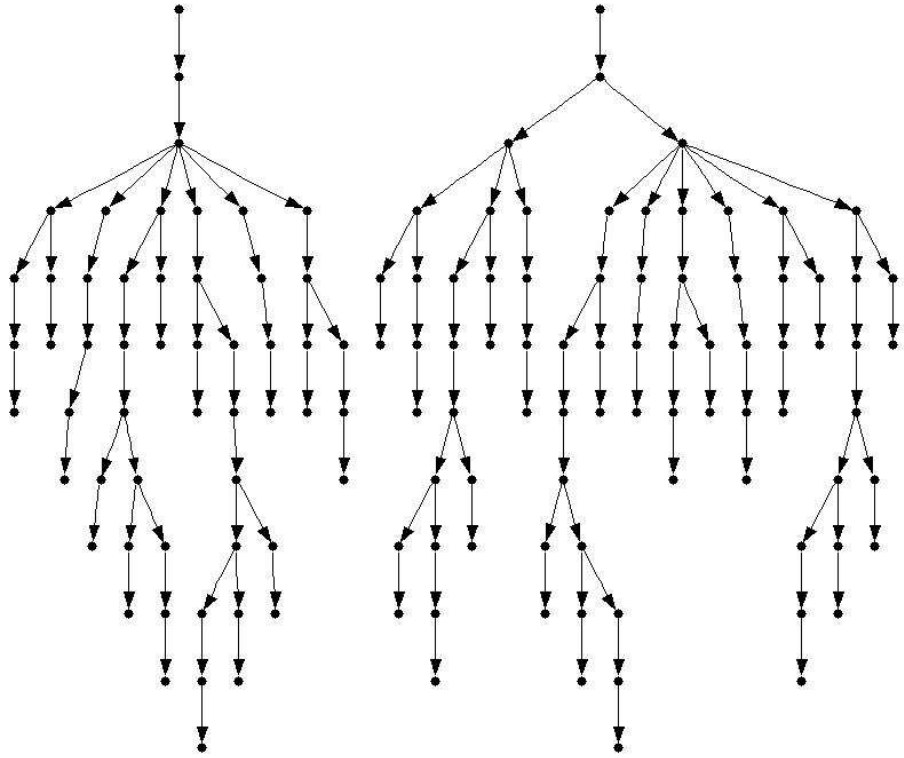


Figure 6.14: OpenRisc abstract parcel automaton

Figure 6.14 shows the abstract operation graph for OpenRisc. There are a total of 35 paths through the operation graph, representing the 35 paths through the pipeline. It may seem surprising that there are so many distinct paths in this processor, but they really do all exist. The paths include all possible combinations of an instruction being fetched into IF0 or IF1, primary and secondary paths through ID and IF, and multi-cycle operations through ALU.

There are 35183 gates in the concrete processor and only 2047 gates after abstraction. The concrete

datapath accounts for 30341 gates and the abstract one for 1458 gates. The abstraction algorithm generated 334 SAT problems. Cumulative SAT solver time is 536 seconds with a maximum of 51.7MB of memory used.

6.7 Summary

Path abstraction performs an implicit DFS like traversal of the concrete parcel automaton using a SAT solver. The algorithm uses propositional formulas called path formulas that stand for equivalence classes of finite path prefixes in the concrete automaton. The algorithm maps a set of equivalent paths of the concrete automaton to an abstract state. Our methodology has been validated with several datapath intensive pipelined designs. The most complex design is a two-wide superscalar OpenRISC microprocessor. There are 35183 gates in the concrete processor and only 2047 gates after abstraction.

Chapter 7

Conclusions

Formal methods are techniques that use mathematical reasoning to prove correctness of hardware and software systems. The main challenges to their wide adoption are automation and capacity. There are two main directions in formal methods that differ with respect to how they represent the behaviour of the system. In the deductive approach the behaviour of the system is described by logic terms that are derived syntactically from the the text of the program. Such methods support reasoning about very general implementations, proving correctness about programs that use unbounded memory or have parameterized implementations with an unbounded number of components. The difficulty in applying deductive techniques is often due to the fact that the underlying logic is not decidable and thus algorithmic approaches are not complete. They instead rely on the user to guide the proof. The other approach in formal verification is to apply automatic reasoning to the finite explicit representation of the program. This representation is often given as a state machine or Kripke structure. Automatic verification is performed by exploring the state space of the system. In this approach the challenge is to overcome the state space explosion problem due to the size of the state space being exponentially larger than the text of the program. In automatic approaches the state explosion problem is alleviated using abstraction and decomposition.

The most complex hardware designs are found in today's microprocessors. Commonly encountered optimizations are out-of-order speculative execution, register renaming and dynamic scheduling. Pipelining is a ubiquitous technique in these designs whereby the execution of instructions is decomposed into a sequence of operations performed at different stages in the pipeline. Pipelining increases the utilization of the circuit and the throughput of instructions, i.e. the rate at which instructions come out of the pipeline. Pipelining brings challenges to both design and verification. Pipeline hazards are the equivalent of race conditions in multi-threaded software. The hazards are denoted by conditions related to the concurrent execution of instructions. Data and control hazards refer to the requirement that the overlapped execution of the instructions in the pipeline have the

same effect as if the instructions were executed one at a time in program order. Structural hazards occur due to resource contention in the pipeline. Their incorrect resolution may lead to resource starvation or deadlocks.

The challenges in the formal verification of pipelined circuits arise in both specification of correctness and verification. The definitions of correctness statements for pipelined circuits, such as the ones based on simulation, must relate a pipeline state to a specification state. Due to pipelining, the implementation variables that correspond to the specification reflect the overlapped updates made by the various instructions in the pipeline. It is therefore not straightforward to relate the implementation with a sequential specification that executes one instruction at a time. Flushing based correctness [Burch and Dill, 1994] mitigates this factor using a pipeline specific form of abstraction called flushing. Flushing transforms an implementation state by completing all in-flight instructions without fetching new ones. Another approach to pipeline correctness is formulated in terms of the correct resolution of hazards [Aagaard, 2003]. This approach is proven to imply the commuting diagram based on flushing. Hazard based correctness has the advantage that correctness can be formulated more easily in terms of pipeline specific behaviour. The state explosion problem in the automatic verification of pipelined circuits arises mainly due to the presence of wide datapaths and memories. Abstraction of memories and datapath is the natural approach to the verification of the control.

Model checking is an automated approach to formal verification. With model checking, correctness properties are defined in temporal logic and verified by state space exploration. To verify systems with large state spaces model checking uses symbolic representations of the state space [Burch et al., 1992], abstraction and decomposition.

Our work is concerned with datapath abstraction for the verification of the control circuitry of pipelined circuits using model checking. Due to the interaction between datapath and control, datapath abstraction must be precise enough so that control properties that are sensitive to the paths and latencies through the pipeline are satisfied by the abstract pipeline.

In Chapter 4 we formalize the pipeline datapath as a state machine called a parcel automaton. The parcel automaton describes the execution of a parcel by the pipeline. It captures the paths parcels take through the pipeline, the transformations that they undergo and the control visible effects they produce. Consequently, the parcel automaton is a representation of the pipeline datapath and its abstraction induces an abstraction of the datapath. Parcel automata allow us to reason about abstractions of the pipeline datapath in terms of simulation and language containment for parcel automata. Conversely, abstract parcel automata can be thought of representing abstract datapaths. Substitution of the concrete datapath by the abstract one induced by the abstract parcel automaton is a form of abstract interpretation. In Chapter 5 we show the soundness of abstraction using parcel automata is proven by showing that simulation and language containment on parcel automata transfer to sim-

ulation and language containment between the concrete pipeline and the abstract one obtained by abstract interpretation.

Chapter 3 describes the model of pipelined circuits as a network of parcel variables and datapath instances through which parcels flow as coordinated by the control circuitry. The variables of the circuit are divided into datapath and control. The separation is enforced by syntactic restrictions on the type of expressions that can be assigned to each of the two kinds of variables. Control variables can be assigned only expressions over control variables. Parcel variables on the other hand, act as output parameters to the datapath instances or can be assigned if-then-else expressions that correspond to mux trees, the leaves of which are parcel variables, constants or non-deterministic choice, and the select signals of the mux nodes are Boolean control expressions. The datapath instances are modeled as combinational circuits with annotations describing the parcel and control variables. The control and the datapath interact through the control input and output variables of the datapaths. Abstract interpretation of the datapath is performed by replacing the concrete datapaths by abstract ones. The type of the pipeline parcel variables is adjusted accordingly. The control is left unchanged.

A parcel represents a group of related values which propagate together during a pipeline computation. Both the values of the parcel and the variables that hold them change during the computation of the pipeline. In a particular pipeline step, the parcel is identified by its variables, which can be register and combinational. We define parcels as non-empty subsets of parcel variables. Parcel automata are labeled transition systems that describe parcel computations. The state of a parcel is an environment over the parcel's registers. The transitions denote the movement of the parcel from the current state variables into the the next-state variables in one pipeline step. The transition label captures the value transformation through the datapaths, the effect on the control variables and the path through the combinational circuitry.

In a pipeline computation multiple parcel computations take place simultaneously. A very important characteristic of the parcel computations that coexist during a computation of the pipeline model, is that within a pipeline step they do not share parcel variables or datapaths. This property of pipeline computations is called parcel independence, or parcel separation and is formalized using parcel maps. Parcel separation is an inductive property that states that the parcel arguments of each datapath belong to the same parcel. Parcel separation implies that the runs of the pipeline model decompose into runs of the parcel automaton. The proof obligations for parcel independence are based on propositional formulas that unfold the pipeline model for one or two consecutive steps. Parcel independence is used to prove Theorem 5.3.1 that states that commuting diagrams between the concrete and abstract parcel automaton states imply a commuting diagram between the containing concrete and abstract pipeline states. Theorem 5.3.1 is used to prove soundness of abstraction using parcel automata for simulation in Theorem 5.4.1 and respectively, for language

containment in Theorem 5.4.8.

The technique for abstracting parcel automata is called path abstraction and is defined in Chapter 6. Path abstraction performs an implicit DFS like traversal of the concrete parcel automaton using a SAT solver. The algorithm uses propositional formulas called path formulas that stand for equivalence classes of finite path prefixes in the concrete automaton. The algorithm maps a set of equivalent paths of the concrete automaton to an abstract state. The abstract state represents all the concrete states that are reachable by concrete paths in the corresponding equivalence class. The main property of this construction is stated in Lemma 6.1.1: the abstract automaton thus defined simulates the concrete parcel automaton. At each iteration, the algorithm finds all extensions of the path formula at the top of the stack that stands for an equivalence class of paths of length k. In the DFS traversal this corresponds to visiting the successors of the currently reached abstract state. If a variable in the domain of a newly created abstract state is in the fan-out of a datapath, it receives a fresh abstract value. Otherwise the variable gets its value from a variable in the previous abstract state. The algorithm terminates if at some point during abstraction new abstract values are no longer created. This corresponds to the concept of terminating paths in the concrete parcel automaton. A path is terminating if it has an infinite suffix in which datapath outputs do not fan-out into next state variables.

Our methodology has been validated with several datapath intensive pipelined designs. The most complex design is a two-wide superscalar OpenRISC microprocessor. There are 35183 gates in the concrete processor and only 2047 gates after abstraction. The concrete datapath accounts for 30341 gates and the abstract one for 1458 gates. The abstraction algorithm generated 334 SAT problems. Cumulative SAT solver time is 536 seconds with a maximum of 51.7MB of memory used.

Bibliography

Mark Aagaard. A hazards-based correctness statement for pipelined circuits. In *CHARME*, pages 66–80, 2003.

Mark D. Aagaard, Vlad C. Ciubotariu, Jason T. Higgins, and Farzad Khalvati. Combining equivalence verification and completion functions. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design*, volume 3312 of *Lecture Notes in Computer Science*, pages 98–112. Springer Berlin / Heidelberg, 2004.

Zaher S. Andraus and Karem A. Sakallah. Automatic abstraction and verification of verilog models. *Design Automation Conference*, 0:218–223, 2004.

Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 19–24, Piscataway, NJ, USA, 2006. IEEE Press. ISBN 0-7803-9451-8.

D.P. Appenzeller and A. Kuehlmann. Formal verification of a powerpc microprocessor. In *Computer Design: VLSI in Computers and Processors, 1995. ICCD '95. Proceedings., 1995 IEEE International Conference on*, pages 79 –84, October 1995.

Ilan Beer, Shoham Ben-David, Daniel Geist, Raanan Gewirtzman, and Michael Yoeli. Methodology and system for practical formal verification of reactive hardware. In David Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 182–193. Springer Berlin / Heidelberg, 1994.

B. Bentley. Validating the Intel® Pentium® 4 microprocessor. In *Design Automation Conference, 2001. Proceedings*, pages 244 – 248, 2001.

Sergey Berezin, Armin Biere, Edmund Clarke, and Yunshan Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 528–528. Springer Berlin / Heidelberg, 1998.

V. Bhagwati and S. Devadas. Automatic verification of pipelined microprocessors. In *Design Automation, 1994. 31st Conference on*, pages 603 – 608, 1994.

Per Bjesse. A practical approach to word level model checking of industrial netlists. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 446–458, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70543-7.

R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677 –691, 1986. ISSN 0018-9340.

Jerry Burch and David Dill. Automatic verification of pipelined microprocessor control. In David Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer Berlin / Heidelberg, 1994.

J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142–170, 1992. ISSN 0890-5401.

Ca Bol Chan, V. Ciubotariu, and M. Aagaard. Pipeline design and verification in Bluenose II. In *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*, pages 1405 –1408, 2007.

Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 241–268. Springer Berlin / Heidelberg, 2002.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8:244–263, April 1986. ISSN 0164-0925.

Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28:626–643, December 1996. ISSN 0360-0300.

Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6:217–232, 1995. ISSN 0925-9856. 10.1007/BF01383968.

E.M. Clarke and R.P. Kurshan. Computer-aided verification. *Spectrum, IEEE*, 33(6):61 –67, June 1996. ISSN 0018-9235.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM*

SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi: http://doi.acm.org/10.1145/512950.512973. URL `http://doi.acm.org/10.1145/512950.512973`.

T.A. Diep and J.P. Shen. Systematic validation of pipeline interlock for superscalar microarchitectures. *Fault-Tolerant Computing, International Symposium on*, 0:0100, 1995.

David L. Dill and John Rushby. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, 29(4):23–24, apr 1996.

D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings., IEEE 1992 International Conference on*, pages 522 –525, October 1992.

N. Een and N. Sorensson. The minisat page. URL `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`.

M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993. ISBN 0-521-44189-7.

Aarti Gupta, Sharad Malik, and Pranav Ashar. Toward formalizing a validation methodology using simulation coverage. *Design Automation Conference*, 0:740, 1997.

Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7:11–19, 1990. ISSN 0740-7459.

A. Hartstein and Thomas R. Puzak. The optimum pipeline depth for a microprocessor. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 7–13, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1605-X.

Jason T. Higgins and Mark D. Aagaard. Simplifying the design and automating the verification of pipelines with structural hazards. *ACM Trans. Des. Autom. Electron. Syst.*, 10:651–672, October 2005. ISSN 1084-4309.

Pei-Hsin Ho, Adrian J. Isles, and Timothy Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 529–536, New York, NY, USA, 1998. ACM. ISBN 1-58113-008-2.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969. ISSN 0001-0782.

Ramin Hojati and Robert K. Brayton. Automatic datapath abstraction in hardware systems. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 98–113, London, UK, 1995. Springer-Verlag. ISBN 3-540-60045-0.

Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In Alan Hu and Moshe Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 122–134. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0028739.

James K. Huggins and David Van Campenhout. Specification and verification of pipelining in the arm2 risc microprocessor. *ACM Trans. Des. Autom. Electron. Syst.*, 3:563–580, October 1998. ISSN 1084-4309.

Hiroaki Iwashita, Satoshi Kowatari, Tsuneo Nakata, and Fumiyasu Hirose. Automatic test program generation for pipelined processors. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, ICCAD '94, pages 580–583, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-89791-690-5.

Christian Jacobi. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. In Ed Brinksma and Kim Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 211–226. Springer Berlin / Heidelberg, 2002.

M. Kaufmann and J.S. Moore. An industrial strength theorem prover for a logic based on common lisp. *Software Engineering, IEEE Transactions on*, 23(4):203 –213, April 1997. ISSN 0098-5589. doi: 10.1109/32.588534.

Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007. ISBN 1932394842.

K. Kohno and N. Matsumoto. A new verification methodology for complex pipeline behavior. In *Design Automation Conference, 2001. Proceedings*, pages 816 – 821, 2001.

Shuvendu Lahiri, Sanjit Seshia, and Randal Bryant. Modeling and verification of out-of-order microprocessors in uclid. In Mark Aagaard and John OLeary, editors, *Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 142–159. Springer Berlin / Heidelberg, 2002.

Leslie Lamport. The hoare logic of concurrent programs. *Acta Informatica*, 14:21–37, 1980. ISSN 0001-5903.

Jeremy Levitt and Kunle Olukotun. Verifying correct pipeline implementation for microprocessors. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, ICCAD '97, pages 162–169, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8200-0.

Panagiotis Manolios and Sudarshan K. Srinivasan. Automatic verification of safety and liveness for xscale-like processor models using web refinements. *Design, Automation and Test in Europe Conference and Exhibition*, 1:10168, 2004. ISSN 1530-1591.

K. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

K. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In Alan Hu and Moshe Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 110–121. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0028738.

K. McMillan. Parameterized verification of the flash cache coherence protocol by compositional model checking. In Tiziana Margaria and Tom Melham, editors, *Correct Hardware Design and Verification Methods*, volume 2144 of *Lecture Notes in Computer Science*, pages 179–195. Springer Berlin / Heidelberg, 2001.

Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, April 1995. ieeecs.

Robin Milner. An algebraic definition of simulation between programs. Technical report, 1971.

P. Mishra, N. Dutt, A. Nicolau, and H. Tomiyama. Automatic verification of in-order execution in microprocessors with fragmented pipelines and multicycle functional units. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '02, pages 36–, Washington, DC, USA, 2002. IEEE Computer Society.

Prabhat Mishra and Nikil Dutt. Graph-based functional test program generation for pipelined processors. In *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, DATE '04, pages 10182–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2085-5-1.

Kedar Namjoshi and Robert Kurshan. Syntactic program transformations for automatic abstraction. In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 435–449. Springer Berlin / Heidelberg, 2000.

C. Norris IP and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996. ISSN 0925-9856. 10.1007/BF00625968.

V. Paruthi, N. Mansouri, and R. Vemuri. Automatic data path abstraction for verification of large scale designs. In *ICCD '98: Proceedings of the International Conference on Computer Design*, page 192, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-9099-2.

Amir Pnueli. The temporal logic of programs. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:46–57, 1977. ISSN 0272-5428.

Jun Sawada and Warren Hunt. Trace table based approach for pipelined microprocessor verification. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 364–375. Springer Berlin / Heidelberg, 1997.

Jens Skakkebak, Robert Jones, and David Dill. Formal verification of out-of-order execution using incremental flushing. In Alan Hu and Moshe Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 98–109. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0028737.

Sudarshan K. Srinivasan and Miroslav N. Velev. Formal verification of an intel xscale processor model with scoreboarding, specialized execution pipelines, and impress data-memory exceptions. *Formal Methods and Models for Co-Design, ACM/IEEE International Conference on*, 0:65, 2003.

Moshe Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham Birtwistle, editors, *Logics for Concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer Berlin / Heidelberg, 1996.

Miroslav Velev. Automatic abstraction of memories in the formal verification of superscalar microprocessors. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 252–267. Springer Berlin / Heidelberg, 2001.

Miroslav N. Velev and Randal E. Bryant. Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 112–117, New York, NY, USA, 2000. ACM. ISBN 1-58113-187-9.

Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, 2003. ISSN 0747-7171.

Phillip Windley and Michael Coe. A correctness model for pipelined microprocessors. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design*, volume 901 of *Lecture Notes in Computer Science*, pages 33–51. Springer Berlin / Heidelberg, 1995.

P.J. Windley. Formal modeling and verification of microprocessors. *Computers, IEEE Transactions on*, 44(1):54 –72, January 1995. ISSN 0018-9340.

F. Zaraket, J. Baumgartner, and A. Aziz. Scalable compositional minimization via static analysis. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 1060–1067, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-9254-X.